NARODOWA AGENCJA
WYMIANY AKADEMICKIEJ

Publikacja finansowana przez NAWA
w ramach działań komplementarnych do tych podejmowanych przez UJD w ramach sojuszu COLOURS

PODSTAWY PROGRAMOWANIA W PYTHONIE

Przykłady i ćwiczenia

Lidia Stępień

Marcin R. Stępień

Hubert Dróżdż

Uniwersytet Jana Długosza w Częstochowie

Podstawy programowania w Pythonie
Przykłady i ćwiczenia

Lidia Stępień, Marcin R. Stępień, Hubert Dróżdż

Częstochowa 2024

Recenzent
dr Marcin Ziółkowski

Redaktor Naczelna Wydawnictwa
Paulina Piasecka-Florczyk

Redaktor statystyczny
Jarosław Kowalski

Korekta
Bożena Woźna-Szcześniak

Redakcja techniczna i projekt okładki
Bożena Woźna-Szcześniak

ISBN 978-83-67984-14-0

Wydawnictwo Naukowe Uniwersytetu Jana Długosza w Częstochowie
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl

Spis treści

Przedmowa

Program komputerowy to szczegółowy zestaw instrukcji określający działania, jakie powinien wykonać komputer. Powstaje on jako wynik procesu tworzenia kodu źródłowego programu w wybranym języku programowania, który jest zbiorem reguł określających, jakie ciągi symboli tworzą program komputerowy oraz jakie obliczenia opisują ten program. Na dalszym etapie przetwarzanie kodu źródłowego programu może odbywać się poprzez:

- *kompilację* - kod źródłowy jest tłumaczony do postaci języka maszynowego;
- *interpretację* - kod źródłowy jest na bieżąco tłumaczony i wykonywany przez dodatkowy program zwany interpreterem.

Wówczas o języku programowania podlegającym kompilacji powiemy, że jest kompilowanym językiem programowania, a o tym podlegającym interpretacji, że jest interpretowanym językiem programowania.

Język Python jest interpretowanym językiem programowania wysokiego poziomu stworzonym przez holenderskiego programistę Guido van Rossuma w 1990 roku. Język ten został nazwany w ślad za programem telewizyjnym BBC „Latający cyrk Monty Pythona". Python rozwijany jest jako projekt *Open Source*, a jego interpretery są dostępne na różne systemy operacyjne. Język Python jest jednym z najmłodszych, ale zarazem najczęściej używanych obecnie języków programowania. Ma dosyć łatwą składnię, stosunkowo niewiele słów kluczowych, a także bardzo bogatą bazę bibliotek, z pomocą których można tworzyć nawet bardzo skomplikowane projekty programistyczne. Język Python ma bogate możliwości zarówno programowania proceduralnego, jak i obiektowego. Jego zaletą jest również to, że słowa kluczowe używane w tym języku są identyczne jak w innych nowoczesnych językach wysokopoziomowych takich jak: C++, JAVA czy PHP. Jednak w porównaniu z tymi językami, tworzenie programów w Pythonie jest bardziej intuicyjne i nie wymaga od początkującego użytkownika szerokiej wiedzy informatycznej i pamiętania wielu edycyjnych szczegółów. Z drugiej strony, w języku tym występują ciekawe rozwiązania, których inne języki nie posiadają, m.in. istnienie obliczeniowego typu liczb zespolonych, operatora potęgowania, domyślnego typu wprowadzanych danych jako danych typu `str` czy bardzo wygodnej struktury danych `list`, a także w zasadzie nieograniczonego zakresu danych liczbowych.

Język Python jest wieloparadygmatowym językiem programowania o wszechstronnych zastosowaniach, zoptymalizowanym pod kątem czytelności kodu, zwięzłej składni i jakości oprogramowania. Python jest obecnie jednym z najpopularniejszych języków programowania. Jest używany w różnych dziedzinach, od tworzenia aplikacji webowych i analizy danych po sztuczną inteligencję i uczenie maszynowe. Duża społeczność i bogactwo bibliotek sprawiają, że Python jest idealny zarówno dla początkujących, jak i zaawansowanych programistów.

W oparciu o wieloletnie doświadczenie dydaktyczne w nauczaniu różnych języków programowania na kierunku informatyka studiów pierwszego stopnia i biorąc pod uwagę różny stopień zaawansowania w programowaniu studentów rozpoczynających studia, można stwierdzić zdecydowanie, że nauka języka Python jako pierwszego języka programowania wydaje się być najlepszym rozwiązaniem. Między innymi dzięki Pythonowi studenci uczą się dbania o czytelność kodu i łatwiej jest im przenieść te umiejętności do innych języków programowania, które nie mają już tak restrykcyjnych wymagań.

Proponowany podręcznik jest praktycznym przewodnikiem skupiającym się na podstawach programowania w języku Python. Nadrzędnym jego celem jest wykształcenie umiejętności programowania w języku Python poprzez omówienie podstawowych konstrukcji programistycznych i struktur danych oraz bogatą ilustrację z wykorzystaniem odpowiednio dobranych przykładów. Ponadto utrwaleniu nabytych umiejętności służyć będą zadania do samodzielnego rozwiązania. Nabyte, poprzez studiowanie tego podręcznika, umiejętności czytelnik będzie mógł wykorzystać w każdym napotkanym systemie oprogramowania opartym na Pythonie.

Niniejszy podręcznik nie mógłby powstać bez nieocenionej pomocy wielu osób, ale na szczególne wyróżnienie zasługuje **dr hab. Andrzej Zbrzezny prof. UJD**. Zawsze mogliśmy liczyć na Jego wsparcie i sugestie będące efektem wieloletniej pracy ze studentami, a także na inspirowanie nas swoimi pomysłami. Dziękujemy, że mieliśmy możliwość czerpania z zasobów tak bogatego doświadczenia naukowego i dydaktycznego.

Rozdział 1

Wprowadzenie do Pythona

W tym rozdziale czytelnik zostanie zapoznany z podstawowymi informacjami o języku Python - od instalacji Pythona w różnych systemach operacyjnych, poprzez pojęcia obiektu, zmiennych, operatorów oraz wyrażeń w Pythonie - aż po pracę z interpreterem.

1.1. Instalacja interpretera Python 3

Pythona w najnowszej wersji można pobrać z odpowiedniej podstrony witryny internetowej tego języka `https://www.python.org`. Szczegóły instalacji Pythona różnią się w zależności od platformy i nie jest naszym celem zagłębianie się w te aspekty. Zainteresowanego czytelnika odsyłamy do wspomnianej wyżej witryny. W tym podręczniku zakładamy, że czytelnik ma już zainstalowanego Pythona (w wersji 3) na swoim komputerze, a dla tych, którzy jeszcze go nie mają, krótko opiszemy sposób instalacji w dwóch najpopularniejszych systemach operacyjnych - Windows i Linux.

W systemach Linux (m.in. w Ubuntu) Python 3 jest standardowym, wbudowanym komponentem tych platform. Natomiast w systemie Arch Linux Pythona 3 instaluje się w terminalu poleceniem: `# python -Sy python3`.

Dla systemu Windows interpreter Pythona 3 można bezpłatnie pobrać ze strony: `https://www.python.org/downloads/windows/`, a następnie wystarczy uruchomić odpowiedni plik w zależności od wersji systemu operacyjnego i jego architektury i na wszystkie pytania odpowiedzieć `Yes` (`Tak`) lub `Next` (`Dalej`). Ważne jest, aby w momencie uruchomienia pliku instalacyjnego, zaznaczyć potrzebę utworzenia zmiennej środowiskowej ze ścieżką do interpretera, aby potem nie wykonywać tego samodzielnie.

1.2. Język Python - podstawy

Programy w języku Python pisze się w plikach tekstowych i zapisuje się je z rozszerzeniem `py`, można więc pracować w zwykłym edytorze tekstu, np. Notatniku. Najwygodniej jest jednak zainstalować zintegrowane środowisko programistyczne (IDE - ang. *Integrated Development Environment*, np. Jupyter, PyCharm, Wings, Visual Studio Code, VIM lub (w Windows) środowisko Python IDLE dostarczane razem z interpreterem podczas instalacji), które zawiera w sobie nie tylko edytor do pisania programów, ale również interpreter, zbiór potrzebnych bibliotek oraz interfejs graficzny, za pomocą którego w łatwy sposób można interpretować, poprawiać i uruchamiać programy.

Python 3.x akceptuje w skryptach znaki pochodzące z zestawu znaków Unicode w systemie `UTF-8`. Program napisany w języku Python jest ciągiem instrukcji. Oznacza to, że ważne są nie tylko instrukcje zawarte w programie, ale także ich kolejność. Instrukcje (słowa kluczowe języka) pisane są małymi literami. Cechą wyróżniającą Pythona spośród innych języków jest stosowanie wcięć do wydzielania bloków kodu źródłowego, co zwiększa jego czytelność i klarowność. Blok musi zawierać co najmniej jedną instrukcję. W przypadku, gdy nie wiemy jeszcze jaki kod powinien znaleźć się w bloku, możemy użyć pustej instrukcji `pass` (lub trzykropka `...`). Python wykrywa początek i koniec bloku na podstawie wcięcia jego instrukcji. Zatem jako blok traktowane są instrukcje wcięte na taką samą głębokość, którą uzyskuje się stosując spacje lub znaki tabulacji, przy czym zgodnie z `PEP8`-Style Guide for Python Code (`https://peps.python.org/pep-0008/`), sugerowane jest użycie 4 spacji zamiast znaku tabulacji.

W Pythonie to wartości, a nie zmienne, posiadają typ, tak więc Python jest językiem z typami dynamicznymi. Każda dana jest reprezentowana przez `obiekt` lub przez relację pomiędzy obiektami. Każdy obiekt ma tożsamość, typ i wartość. Od chwili utworzenia obiektu jego tożsamość nigdy się nie zmienia. O tożsamości obiektu można myśleć jak o adresie obiektu w pamięci. Do porównania tożsamości dwóch obiektów służy operator `is`. Wbudowana funkcja `id` zwraca wartość całkowitą, reprezentującą tożsamość obiektu - w standardowej implementacji funkcja ta zwraca adres obiektu, przekształcony do postaci liczbowej. Typ określa zbiór atrybutów i operacji, które można wykonać na obiekcie oraz definiuje zbiór dopuszczalnych wartości obiektu.Typ obiektu, podobnie jak jego tożsamość, również nie może ulec zmianie. Typ obiektu, który również jest obiektem, można pobrać za pomocą wbudowanej funkcji `type`.

W języku Python mamy do czynienia z następującymi, wbudowanymi typami danych:

1. `int` - liczby typu całkowitego, np. `-19, 0, 1` itd., można używać znaku podkreślenia `_` jako separatora (co trzy cyfry) w dużych liczbach,
2. `float` - liczby zmiennoprzecinkowe (rzeczywiste), np. `3.14, -44.99` itd.,
3. `complex` - liczby zespolone, np. `1+4j, -2j, -39-34j` itd.,

4. `str` - łańcuchy znaków, napisy (np. słowa) - jest to domyślny typ wprowadzanych danych, np. `"Ala", 'Ola',''` itp.,
5. `bool` - wartości logiczne - zawiera tylko dwie wartości `True`(1) oraz `False`(0).

Zmienna w Pythonie jest nazwą, która jest odniesieniem do obiektu. Nazwa zmiennej to identyfikator rozpoczynający się od znaku podkreślenia lub litery plus dowolnej liczby liter, cyfr lub podkreśleń, przy czym wielkość liter jest rozróżnialna. Wymienione niżej słowa kluczowe są zastrzeżone i nie mogą być użyte jako nazwy zmiennych:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Funkcjonują różne konwencje nazywania zmiennych, ale za dobrą praktykę uznaje się stosowanie `snake_case`, w której nazwy definiujemy przy użyciu małych liter, a słowa oddzielamy znakiem podkreślenia `_`. Gdy mamy kilka zmiennych o podobnym znaczeniu, na końcu dodajemy liczby, np. `name1, name2`. Nazwa zmiennej powinna wyrażać jej przeznaczenie. Poprawnym jest stosowanie nazw anglojęzycznych (zawsze w rozwiązaniach komercyjnych), a w przypadku stosowania nazw polskich - nie powinno się stosować polskich znaków diakrytycznych.

Aby utworzyć zmienną oraz nadać jej wartość należy użyć instrukcji przypisania postaci:

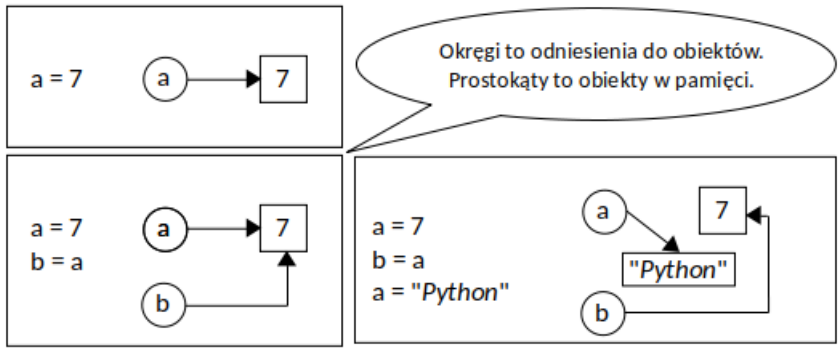
```
zmienna = wyrażenie
```

Do zmiennej `zmienna` przypisywane jest odniesienie do obiektu utworzonego w wyniku obliczenia wartości wyrażenia `wyrażenie`. Przypisanie do zmiennej nie jest wypisywane przez interpreter.

Na rysunku 1.1 utworzono zmienną o nazwie `a`, która jest odniesieniem do obiektu w pamięci, który przechowuje wartość 7. Następnie tworzona jest druga zmienna o nazwie `b`, która poprzez przypisanie `b = a` jest odniesieniem do tego samego obszaru pamięci, który wskazuje zmienna `a`. W momencie przypisania do zmiennej `a` nowej wartości, tworzony jest w pamięci obiekt przechowujący tę wartość, z którym zmienna `a` zostaje powiązana. Z obiektem przechowującym wartość 7 pozostaje powiązana już tylko zmienna `b`.

W języku Python możliwe jest przypisanie wartości do wielu zmiennych jednocześnie. Przykładowo, jeśli chcielibyśmy, aby zmienne `a, b, c, d` miały jednocześnie wartość 12, możemy to zrobić albo poprzez cztery odrębne przypisania, albo możemy użyć pojedynczej instrukcji:

```
a = b = c = d = 12
```



Rysunek 1.1: Odniesienia do obiektów oraz obiekty w Pythonie

Jeśli chcielibyśmy określić różne wartości dla `a` i `b`, to również można to zrobić w jednej linii. W takim przypadku najpierw określamy zmienne, do jakich chcemy przypisać wartości i oddzielamy je przecinkami, stawiamy znak równości i określamy kolejno ich wartości, np. `a, b, c = 1, 2, 3` Jednym z zastosowań tego zapisu jest zamiana wartości tak, aby `a` przyjęło wartość zmiennej `b` i na odwrót: `a, b = b, a`

W Pythonie wyrażenia budowane są przy pomocy:

- Operatorów arytmetycznych:
  - dodawania + i odejmowania −,
  - mnożenia ∗ i dzielenia /,
  - części całkowitej z dzielenia // i reszty z dzielenia %,
  - potęgowania ∗∗.
- Operatorów porównania i logicznych, gdzie:
  - operatory porównania:
    == - równe,
    ! = - różne,
    < - mniejszy,
    > - większy,
    <= - mniejszy lub równy,
    >= - większy lub równy;
  - operatory logiczne w kolejności malejącej priorytetów:
    `not` (negacja - zaprzeczenie),
    `and` (koniunkcja - "i") oraz
    `or` (alternatywa - "lub").

W Pythonie operatory mają określony priorytet:

- Nawiasy mają najwyższy priorytet. Są używane do wymuszenia obliczania wartości wyrażenia w żądanej kolejności.

- Potęgowanie ma kolejny najwyższy priorytet.
- Mnożenie, dzielenie, dzielenie całkowite i reszta z dzielenia mają ten sam priorytet, który jest wyższy niż dodawanie i odejmowanie, które również mają ten sam priorytet.
- Operatory o tym samym priorytecie są obliczane od lewej do prawej. W algebrze mówimy, że są lewostronnie łączne.
- Wyjątkiem jest operator potęgowania, który jest prawostronnie łączny (czyli jest obliczany od prawej do lewej).

Operatory z operandami typów mieszanych przekształcają:

- operandy boolowskie w całkowite, zmiennoprzecinkowe lub zespolone,
- operandy całkowite w zmiennoprzecinkowe lub zespolone,
- operandy zmiennoprzecinkowe w zespolone.

W Pythonie można korzystać również ze specjalnej, opcjonalnej formy przypisania, która w praktyce często przydaje się programistom. Jest to przypisanie rozszerzone, które jest skrótem łączącym wyrażenie i przypisanie w zwięzły sposób. Przykładowo przypisanie `i += 1` ma ten sam efekt co `i = i + 1`. Zwróć uwagę, że przypisanie rozszerzone może być zastosowane względem zmiennej tylko wtedy, gdy występuje ona po obydwu stronach przypisania. W przypisaniu rozszerzonym w wyrażeniach można korzystać ze wszystkich wymienionych wcześniej operatorów arytmetycznych.

W logice klasycznej każde zdanie może przyjąć tylko jedną z dwóch wartości logicznych: `prawda` - `1` lub `fałsz` - `0`. W Pythonie typ `bool` jest podtypem typu `int`, a zatem wywołanie funkcji wbudowanej `issubclass`(`bool`, `int`) zwróci wartość `True`:

```
>>> issubclass(bool, int)
True
>>> issubclass(int, int)
True
>>> issubclass(int, float)
False
```

W prawie wszystkich kontekstach wartości boolowskie `False` i `True` zachowują się jak wartości `0` i `1`, odpowiednio. Wyjątkiem jest to, że po przekonwertowaniu wartości boolowskiej na łańcuch, zwracany jest łańcuch `'False'` lub `'True'`. W Pythonie każdy obiekt może być w razie potrzeby potraktowany jako wartość boolowska `True`. Następujące obiekty są traktowane jako wartość `False`:

- `None` - specjalna wartość, która symbolizuje brak wartości, wartość nieokreśloną;
- `False`;
- zero z każdego typu numerycznego, przykładowo: `0, 0.0, 0j`;
- dowolna pusta sekwencja, przykładowo: `''`, `()`, `[]`;
- dowolne puste odwzorowanie, przykładowo: `{}`;

- instancje klas zdefiniowanych przez użytkownika, o ile w klasie zdefiniowana jest metoda `__bool__()` lub metoda `__len__()` oraz metody te zwracają dla tych instancji wartość boolowską `False` lub liczbę całkowitą zero.

Wszystkie pozostałe wartości traktowane są jak wartość boolowska `True`.

Funkcja wbudowana `bool` zwraca dla każdego obiektu jego wartość logiczną, np.

```
>> bool(None)
False
>> bool(6)
True
>> bool("")
False
>> bool("Python. Hello World!")
True
>> bool([])
False
>> bool([10, 20, 30, 40])
True
```

W Pythonie można łączyć ze sobą porównania, jak robi się to w matematyce, tworząc jedno wyrażenie postaci:

$$a < b < c < d$$

nazywane porównaniem kaskadowym. Jest ono równoważne wyrażeniu:

$$a < b \text{ and } b < c \text{ and } c < d$$

Porównania kaskadowe mogą mieć dowolną długość, lecz niejawna obecność spójnika `and` oznacza, że obliczenia zatrzymają się natychmiast po stwierdzeniu tego, czy prawdziwość całego wyrażenia może być jednoznacznie obliczona, np. w wyrażeniu

$$a < b < c < d$$

jeżeli podwyrażenie $a < b$ jest fałszywe, to całe wyrażenie jest fałszywe i pozostałe porównania nie będą już wartościowane.

Niezbędnym elementem tworzenia kodu w dowolnym języku programowania jest pisanie komentarzy. W języku Python wyróżniamy następujące komentarze:

- Komentarz jednowierszowy rozpoczynający się od znaku `#`, np.

```
1 # komentarz jednolinijkowy
2 # a = 1
```

- Komentarz wielowierszowy - tekst znajdujący się pomiędzy potrójnym znakiem apostrofu `'''` lub potrójnym znakiem cudzysłowu `"""`.

```
'''
komentarz wielolinijkowy z użyciem apostrofu
a = 1
b = a
'''

"""
komentarz wielolinijkowy z użyciem cudzysłowia
a = 1
b = a
"""
```

Do wypisywania wartości argumentów na standardowym wyjściu (ekranie) służy funkcja `print`. Jej argumentami mogą być napisy, literały, struktury danych lub nazwy zmiennych. Dodatkowo obsługuje argumenty ze słowami kluczowymi pozwalające na zastosowanie specjalnych trybów wyświetlania. Domyślnie funkcja `print` wstawia pomiędzy wyświetlane argumenty separator będący spacją, a na końcu wstawia znak nowej linii. Stąd wywołanie tej funkcji bez argumentów - `print()` - powoduje wstawienie pustej linii na ekran. Można zmienić te ustawienia przypisując parametrom `sep` oraz `end` nowe wartości.

LISTING 1.1: *Użycie funkcji* `print`

```
print("I LOVE PROGRAMMING IN PYTHON")
print("I LOVE PROGRAMMING IN PYTHON", end='')
print("I LOVE PROGRAMMING IN PYTHON")
```

Na listingu 1.1 przedstawiamy program, który wypisuje na ekranie trzy takie same komunikaty przy pomocy funkcji `print`, przy czym druga z nich dodatkowo zmienia domyślny sposób wstawiania po wypisanym komunikacie znaku końca linii na znak pusty. Powoduje to pojawienie się na ekranie kolejnego komunikatu rozpoczynającego się bezpośrednio za ostatnim znakiem komunikatu poprzedzającego go.

Funkcja `input` zwraca wartość wczytaną ze standardowego wejścia, czyli klawiatury, domyślnie w postaci łańcucha znaków. Argumentem jest napis (`str`), który wyświetli się na ekranie użytkownika, co ilustruje kod z listingu 1.2.

LISTING 1.2: *Użycie funkcji* `input`

```
a = input("Podaj liczbę całkowitą: ")
```

Aby wczytana wartość była interpretowana jako typ inny od domyślnego `str`, należy dokonać jej konwersji na odpowiedni typ, co pokazano na listingach 1.3 oraz 1.4, konwertując wczytaną wartość na liczbę całkowitą i rzeczywistą, odpowiednio.

Listing 1.3: *Wprowadzanie liczb całkowitych*

```
1 a = int(input("Podaj liczbę całkowitą: "))
```

Listing 1.4: *Wprowadzanie liczb zmiennoprzecinkowych*

```
1 b = float(input("Podaj liczbę rzeczywistą: "))
```

Listing 1.5: *Wypisywanie wyników obliczeń*

```
1 a = int(input("Podaj pierwszą liczbę: "))
2 # Podaj pierwszą liczbę: 12
3 b = int(input("Podaj drugą liczbę: "))
4 # Podaj drugą liczbę: -33
5 print("Suma liczby",a,"oraz",b,"wynosi",a+b, sep='_')
6 # Suma liczby_12_oraz_-33_wynosi_-21
7 print(a,"+",b,"=", a+b)
8 # 12 + -33 = -21
```

Program z listingu 1.5 prosi użytkownika o podanie dwóch liczb całkowitych, a następnie wypisuje na ekranie te liczby oraz ich sumę. Robi to na dwa sposoby. W linii 5, słownie z użyciem znaku podkreślenia jako separatora, a w linii 7 zgodnie z matematycznym zapisem. Kolejność argumentów w funkcji `print` jest zgodna z kolejnością pojawiających się na ekranie informacji. Zmienne oddzielone są od napisów przecinkiem i oznaczają wstawienie w ich miejsce wartości, do których się odnoszą. Również obliczana suma jest wyrażeniem arytmetycznym oddzielonym przecinkiem. Zanim zostanie wypisana na ekranie suma wskazanych zmiennych, zostaną pobrane ich wartości i obliczona wartość wyrażenia. Ponieważ domyślnym separatorem pomiędzy argumentami w funkcji `print` jest spacja, nie wstawiamy w obrębie napisów dodatkowych spacji. W komentarzach umieszczono informacje, które w trakcie wykonywania programu pojawiają się na ekranie. W liniach pierwszej i trzeciej użytkownik wprowadził przykładowe liczby całkowite i zatwierdził je klawiszem `Enter`.

1.3. Praca z interpreterem

Z Pythonem możemy pracować na dwa sposoby. Obok interpretacji skryptów, mamy również możliwość korzystania z trybu interaktywnego. Takie rozwiązanie wydaje się bardzo użyteczne w przypadku, gdy szybko chcemy przetestować działanie wprowadzonego, najczęściej niezbyt obszernego kodu, sprawdzić działanie pewnych funkcji czy poleceń lub wykorzystać interpreter jako zwykły kalkulator, co ma miejsce w bardzo wielu przypadkach. Używanie trybu interaktywnego ma niestety istotną wadę. Polecenia wpisywane w ten sposób nie są przechowywane, w związku z tym nie mogą zostać wykonane kolejny raz bez ponownego ich wprowadzenia.

Uruchomienie interaktywnej sesji Pythona różni się odrobinę w zależności od platformy, na której pracujemy. W systemie Linux wystarczy, że w powłoce systemowej wydamy polecenie `Python3`. Na początku zostanie wyświetlona wiadomość powitalna zawierająca numer wersji Pythona, z którym aktualnie pracujemy oraz informacje o prawach autorskich. Następnie, pojawi się znak zachęty, który zwykle jest zestawem trzech znaków większości (`>>>`). Oznacza to, że Python oczekuje na interakcję z użytkownikiem. W przypadku, gdy wprowadzana jest wielolinijkowa instrukcja, Python, oczekując na pozostałe linie, zmienia znak zachęty na trzy kropki (`...`). Aby opuścić sesję, należy użyć sekwencji klawiszy `Ctrl+D` (w przypadku systemu Linux) lub `Ctrl+Z` (w przypadku Windows). Oto przykładowy wygląd wiadomości wyświetlanych na początku pracy z trybem interaktywnym:

```
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Przedstawimy teraz przykłady wykorzystania trybu interaktywnego. Na początku spróbujmy użyć go jako kalkulatora.

```
>>> 2 + 3           # obliczenie sumy liczb 2 i 3
5
>>> 3 * 4           # obliczenie iloczynu liczb 3 i 4
12
>>> 2 ** 3          # wyznaczenie trzeciej potęgi liczby 2
8
>>> 5 / 2           # wyznaczenie ilorazu liczb 5 i 2
2.5
>>> 2 ** 0.5        # obliczenie pierwiastka kwadratowego liczby 2
1.4142135623730951
>>> (30 + 3*4) / 2  # wynikiem działania będzie liczba typu float
21.0                # (domyślny typ w przypadku operatora dzielenia)
>>> 2 * 3.25 - 3    # przykład wykonania operacji na danych
3.5                 # różnych typów
>>> 1_000_000 * 2   # obliczenie iloczynu liczb 1000000 i 2
2000000             # znak '_' jest separatorem cyfr używanym
                    # dla poprawienia czytelności dużych liczb
>>> 0b1110          # wyświetlenie dziesiętnej wartości liczby
14                  # przedstawionej w systemie dwójkowym
>>> 0b10            # analogicznie jak wyżej
2
```

```
>>> 0b1110 + 0b10   # obliczenie sumy dwóch liczb binarnych
16                  # wynik jest w systemie dziesiętnym
```

Spróbujmy teraz wykonać kolejne obliczenia, ale tym razem z użyciem zmiennych. Zmiennej `cena` przypiszemy kwotę netto danego towaru, a w zmiennej `vat` będziemy przechowywać stawkę podatku VAT. Posługując się trybem interaktywnym wyznaczymy cenę brutto. Wykorzystamy również znak podkreślenia, którego użycie w tym przypadku będzie skutkowało podstawieniem wyniku ostatnio wyświetlanego wyrażenia.

```
>>> vat = 0.23
>>> vat             # możemy w każdej chwili sprawdzić wartość
0.23                # zmiennej vat
>>> cena = 55
>>> cena            # podobnie w przypadku zmiennej cena, możemy
55                  # wyświetlić jej wartość
>>> cena * vat
12.65
>>> cena + _
67.65
```

Może się również zdarzyć, że w sesji interaktywnej popełnimy błąd, na przykład próbując wyświetlić wartość zmiennej przed przypisaniem do niej wartości. W takim przypadku otrzymamy stosowny komunikat o błędzie:

```
>>> nowa_cena
Traceback (most recent call last):
    File ‘‘<stdin>’’, line 1, in <module>
NameError: name ‘nowa_cena’ is not defined
```

Jak już zauważyliśmy, w trybie interaktywnym, w odróżnieniu do zapisywania programów w plikach, nie musieliśmy używać instrukcji `print` do wyświetlania wyników wyrażeń, a jedynie umieścić samą nazwę zmiennej. Należy jednak pamiętać, że nie zawsze możemy zamiennie stosować te dwie konwencje. Popatrzmy na poniższy przykład:

```
>>> napis = ’Py\nthon’
>>> napis
’Py\nthon’
>>> print(napis)
Py
thon
```

Jak widzimy, funkcja `print` usuwa znaki cytowania oraz przetwarza znaki specjalne.

Wspominaliśmy już o możliwości wprowadzania poleceń wielowierszowych. Spróbujmy więc wyświetlić na ekranie sześć linii, z których każda składa się z pięciu symboli `"@"`.

```
>>> for _ in range(6):
...     print(''@''*5)    # naciskamy klawisz Enter
...                       # w tym miejscu można wprowadzić kolejne
@@@@@                     # instrukcje pętli lub zakończyć instrukcję
@@@@@                     # wielowierszową poprzez wprowadzenie
@@@@@                     # pustego wiersza, czyli poprzez ponowne
@@@@@                     # naciśnięcie klawisza Enter
@@@@@
@@@@@
```

Zwróćmy uwagę na zmianę znaku zachęty na "..." począwszy od drugiej linii kodu. Pamiętajmy także o zachowaniu odpowiedniego wcięcia w przypadku instrukcji, które mają zostać wykonane w pętli. Zakończenie instrukcji wielolinijkowej sygnalizujemy wprowadzeniem pustej linii (ponownym naciśnięciem klawisza Enter). Poniżej przedstawiamy jeszcze jeden przykład, tym razem wyświetlimy w jednej linii liczby:

```
>>> a = 5
>>> while a > 0:
...     print(a,end=' ')
...     a -= 1
...
5 4 3 2 1 >>>
```

Szczegółowe informacje o pętlach czytelnik odnajdzie w podrozdziale 2.3.

1.4. Moduły

Program Pythona często składa się z kilku plików tekstowych zawierających kod napisany w tym języku, z których jeden jest głównym plikiem `najwyższego poziomu`, a pozostałe to pliki pomocnicze - tzw. `moduły`, z których każdy jest przestrzenią nazw swoich atrybutów. Plik najwyższego poziomu zawiera instrukcje sterowania programu wykorzystujące narzędzia zdefiniowane w plikach modułów, które z kolei mogą korzystać z narzędzi zdefiniowanych w innych modułach.

Moduł możemy zaimportować na dwa sposoby poleceniami:

- `import` - które wymaga podania nazwy modułu względem nazw w nim zdefiniowanych (np. `modul.nazwa`) lub
- `from` - które dodatkowo kopiuje jedną lub większą liczbę zmiennych z importowanego modułu do zasięgu, w którym się pojawia.

Sposób importowania pokażemy na przykładzie biblioteki matematycznej `math`, z której zaimportujemy funkcję `sin` na dwa sposoby:

- używając `import`:

```
import math
print(math.sin(12))
# -0.5365729180004349
```

- używając `from`:

```
from math import sin
print(sin(12))
# -0.5365729180004349
```

Importowanie modułu składa się z trzech podstawowych kroków:

1. Odszukania pliku modułu.
2. Skompilowania go do postaci kodu bajtowego (o ile jest to konieczne).
3. Wykonania kodu modułu w celu utworzenia definiowanych przez niego obiektów.

W czasie działania programu trzy powyższe kroki są wykonywane jedynie przy `pierwszym` importowaniu modułu. Późniejsze importy pobierają już tylko załadowany obiekt modułu z pamięci (`sys.modules`). Jeśli w tabeli `sys.modules` nie ma obiektu modułu, rozpoczyna się proces importowania składający się z wyżej wymienionych trzech kroków.

Najistotniejszą częścią procedury importowania jest lokalizacja pliku, który ma być zaimportowany. Ścieżka wyszukiwania modułów Pythona składa się z zestawienia następujących komponentów:

1. Katalog główny programu (automatycznie) - Python najpierw szuka importowanych plików w katalogu głównym. Katalog główny, w zależności od sposobu uruchamiania kodu, będzie katalogiem zawierającym uruchamiany plik skryptowy najwyższego poziomu tego programu lub katalog, w którym pracujemy (aktualny katalog roboczy), w przypadku pracy interaktywnej. Wszystkie operacje importowania będą zatem działać automatycznie. Istnieje ryzyko przypadkowego ukrycia modułów biblioteki, np. w przypadku użycia tej samej nazwy dla pliku najwyższego poziomu, co nazwa importowanego modułu.
2. Katalogi `PYTHONPATH` (konfigurowalne) - w kolejnym kroku Python przeszukuje katalogi podane w zmiennej środowiskowej `PYTHONPATH`, od lewej do prawej strony, o ile ją ustaliliśmy, bo zmienna ta nie jest predefiniowana. W `PYTHONPATH` podaje się listę zdefiniowanych przez użytkownika i specyficznych dla platformy nazw katalogów zawierających pliki z kodem Pythona. Ustawienie to ma znaczenie jedynie wtedy, gdy importuje się pliki pomiędzy różnymi katalogami.
3. Katalogi biblioteki standardowej (automatycznie).
4. Zawartość wszystkich plików `.pth` (konfigurowalne) - Python pozwala użytkownikom dodawać katalogi do ścieżki wyszukiwania modułów przez umieszczenie ich

po jednym w wierszu w pliku tekstowym kończącym się rozszerzeniem `.pth` (ang. *path*). Jest to zaawansowana opcja instalacyjna stanowiąca alternatywę dla ustawień zmiennej środowiskowej `PYTHONPATH`. Więcej informacji można odnaleźć w dokumentacji biblioteki standardowej Pythona - w szczególności modułu `site` pozwalającego na konfigurację lokalizacji bibliotek Pythona oraz plików ścieżek.

5. Katalog `site-packages` dla zewnętrznych pakietów rozszerzeń (automatycznie) - Python dodaje podkatalog `site-packages` swojej biblioteki standardowej do ścieżki wyszukiwania modułów. Umownie jest to miejsce, w którym instaluje się większość rozszerzeń innych firm, często automatycznie przez narzędzie `distutils`. Ponieważ ich katalog instalacyjny jest zawsze częścią ścieżki wyszukiwania modułów, klienty mogą importować moduły takich rozszerzeń bez konieczności osobnego ustawiania ścieżki.

Po zaimportowaniu modułu `sys` biblioteki standardowej, można wyświetlić wbudowaną listę `sys.path` i zobaczyć jak skonfigurowana jest ścieżka wyszukiwania modułów na komputerze, np.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10',
'/usr/lib/python3.10/lib-dynload',
'/usr/local/lib/python3.10/dist-packages',
'/usr/lib/python3/dist-packages']
```

Lista ta zapewnia ręczne dostosowywanie ścieżek wyszukiwania dla skryptów metodą `sys.path.append` lub `sys.path.insert` - przy czym w zmienionej postaci przetrwa tylko jedno uruchomienie programu.

Poza nazwą modułu w operacji importowania można wymienić również ścieżkę katalogu. Katalog z kodem Pythona nazywa się `pakietem`, stąd `importowanie pakietów` (ang. *package import*). Importowanie pakietów polega na umieszczeniu w instrukcji `import` (`from`) zamiast nazwy pliku (modułu) ścieżki nazw rozdzielonych od siebie kropkami:

```
1 import dir1.dir2.nazwa_modulu
2 from dir1.dir2.nazwa_modulu import x
```

Ścieżka z kropkami odpowiada ścieżce w systemie plików prowadzącej do pliku o nazwie `nazwa_modulu.py` (lub podobnego).

Każdy katalog wymieniony w ścieżce instrukcji importowania pakietu może (do wersji 3.3 Pythona musi) zawierać plik o nazwie `__init__.py`, w przeciwnym razie operacja importowania zakończy się niepowodzeniem. Przy czym katalog nadrzędny nie musi zawierać tego pliku, ponieważ nie jest on wymieniony w samej instrukcji `import`, a sam plik `__init__.py` może być pusty.

Jeżeli struktura katalogów wygląda następująco:

```
dir0/dir1/dir2/nazwa_modulu.py
```

a instrukcja `import` ma postać:

```
import dir1.dir2.nazwa_modulu
```

to zastosowanie mają poniższe reguły:

- katalog `dir1` oraz `dir2` muszą zawierać plik `__init__.py`;
- katalog nadrzędny `dir0` nie musi zawierać pliku `__init__.py`; jeżeli plik ten będzie się w nim znajdował, zostanie zignorowany;
- katalog `dir0`, a nie `dir0/dir1`, musi być umieszczony w ścieżce wyszukiwania modułów `sys.path`.

Pliki `__init__.py` mogą zawierać kod Pythona, który uruchamiany będzie automatycznie przy pierwszym zaimportowaniu katalogu przez program i aktywuje inicjalizacje wymagane przez pakiet.

Rozważmy następującą strukturę katalogów:

```
/home/uzytkownik/Pulpit/
  py3/              # katalog w ścieżce wyszukiwania modułów
    __init__.py
    dir1/
      __init__.py
      dir2/
        __init__.py
        funkcje.py
```

Plik `funkcje.py` zawiera funkcje definiujące trzy podstawowe operacje arytmetyczne:

```
# funkcje.py
def suma(a,b):
  return a + b
def iloczyn(a,b):
  return a * b
def roznica(a,b):
  return a - b
```

Tworzymy skrypt `moduly.py` w katalogu: `/home/uzytkownik/Pulpit/Python/programy` o zawartości:

```
import sys
sys.path.append('/home/lidka/Pulpit/py3')
import dic1.dic2.funkcje as m
print(m.suma(1,2))
```

Przy pierwszym imporcie za pomocą katalogu, Python automatycznie wykonuje cały kod z pliku `__init__.py` tego katalogu. Zatem pliki te mogą być miejscem do wstawienia kodu inicjalizującego stan wymagany przez pakiet, np. utworzenie wymaganych plików lub otwarcie połączenia z bazą danych. Dzięki `__init__.py` deklarujemy, że dany katalog jest pakietem Pythona. Bez tego zabezpieczenia Python mógłby wybrać katalog, który nie ma nic wspólnego z kodem programu, tylko dlatego, że pojawia się wcześniej w ścieżce wyszukiwania. Pakiety przestrzeni nazw Pythona 3.3 w znacznym stopniu redukują tę rolę, gdyż uzyskują podobny efekt w sposób algorytmiczny, skanując ścieżkę w poszukiwaniu kolejnych plików. W modelu importowania pakietów ścieżki katalogów stają się po zaimportowaniu prawdziwymi ścieżkami zagnieżdżonych obiektów. Po zaimportowaniu przykładowo wyrażenie `dict1.dict2.funkcje` działa i zwraca obiekt modułu, którego przestrzeń nazw zawiera wszystkie zmienne przypisane przez plik `__init__.py` katalogu `dict2`. Po szczegółowy opis pakietów przestrzeni nazw zainteresowanego czytelnika odsyłamy do fachowej literatury wskazanej w bibliografii lub dokumentacji Pythona.

Instrukcja `from` importuje cały moduł, podobnie do instrukcji `import`, jednak dodatkowo kopiuje jeszcze jedną lub większą liczbę zmiennych z importowanego modułu do zasięgu, w którym się pojawia. Pozwala to na bezpośrednie używanie zaimportowanych zmiennych bez konieczności podawania w wyrażeniu nazwy modułu. Instrukcja `from` może uszkadzać przestrzenie nazw nadpisując zmienne wykorzystywane w zakresie bieżącym. Forma `from ... import *` jest z tego względu jeszcze bardziej niebezpieczna.

Przy tworzeniu kodu programu operującego na modułach, często przydatna jest funkcja `reload`, która wymusza ponowne zaimportowanie modułu w sytuacji, gdy chcemy w czasie programowania uzyskać nowszą wersję kodu źródłowego modułu lub gdy aplikacja wymaga dynamicznego dostosowania do potrzeb użytkownika. W przypadku użycia funkcji `reload` wraz z `from`, ta ostatnia może powodować problemy, np. gdy zmienna będzie odnosić się do poprzedniej wersji obiektów.

Czytelnika zainteresowanego poszerzeniem informacji o sposobach importowania modułów w Pythonie zachęcamy do wnikliwego przestudiowania dokumentacji tego języka.

1.5. Zadania do samodzielnego rozwiązania

1. Pamiętając o tym, że symbol `_` przechowuje ostatnią wydrukowaną wartość, oblicz wartość wyrażenia: `((12**2 - 23)*15)/4`.
2. Przypisz zmiennej `x` wartości różnych typów: łańcuch znaków, liczba całkowita, liczba rzeczywista i sprawdź typ zmiennej używając funkcji `type`.
3. Wprowadź różne wartości dla zmiennej `x` używając instrukcji `input`, sprawdzając za każdym razem jakiego jest typu. Co należy zrobić, aby typ zmiennej był zgodny z typem wprowadzanej wartości?

4. Wczytaj wartości dwóch liczb całkowitych `a`, `b` i wypisz na ekranie:
   (a) sumę,
   (b) różnicę,
   (c) iloczyn,
   (d) iloraz,
   (e) część całkowitą z dzielenia `a` przez `b`,
   (f) resztę z dzielenia `b` przez `a`,
   (g) `b`-tą potęgę liczby `a`.
5. Przetestuj co się stanie, gdy wpiszemy wyrażenie:
   (a) `1/0`,
   (b) `print(x)`, gdzie `x` nie był wcześniej używany?
6. Sprawdź, jak działa mnożenie tekstu przez liczbę całkowitą. Pomnóż napis `'Nie ważne jak się zaczyna, ważne jak się kończy ;)'` razy `100`.
7. Oblicz wartość wyrażenia: $3 + \frac{2^2}{5 \cdot 4}$.
8. Oblicz wartość wyrażenia: $\sqrt{16} + 2^3$.
9. Wykorzystaj operator ** do obliczenia wartości wyrażenia $\sqrt{2} + \sqrt[4]{2} + 2^3$.
10. Sprawdź jaka jest różnica w wyniku wykonania dzielenia `5/2` oraz `5//2`.
11. Wykonaj następujące polecenia i opisz efekt każdego z nich:

```
a = 10
a += 10
a = a + 10
print('a')
print(a)
print(a + 10)
```

12. Sprawdź wynik działania polecenia `123_000_000 + 123`. Do czego służy znak podkreślenia (_) występujący w liczbie?
13. Jaka jest różnica w działaniu poleceń: `3/3*3` oraz `3/(3*3)`. Odpowiedź uzasadnij.
14. Oblicz -3^2 oraz $(-3)^2$.
15. Na podstawie wprowadzonej kwoty brutto oblicz kwotę netto (wartość podatku możesz zdefiniować dowolnie, np. `VAT = 0.23` lub `VAT = 0.08`).
16. Wprowadź wartości zmiennych `waga` (w kilogramach) i `wzrost` (w metrach), a następnie na podstawie tych danych oblicz wskaźnik masy ciała $BMI = \frac{waga}{wzrtost^2}$(BMI - ang. *Body Mass Index*).

Rozdział 2

Podstawowe konstrukcje programistyczne

W tym rozdziale czytelnik zostanie zapoznany z podstawowymi konstrukcjami programistycznymi: instrukcją warunkową oraz instrukcjami iteracyjnymi.

2.1. Instrukcja warunkowa

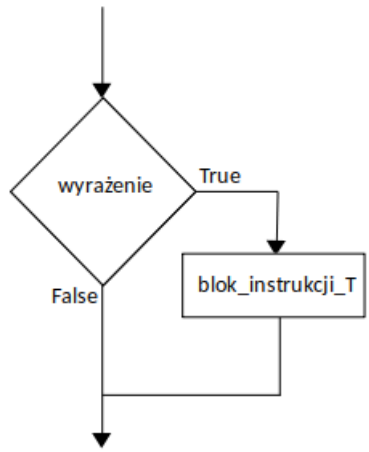
Instrukcja warunkowa `if` w Pythonie jest podstawowym narzędziem wyboru, z pomocą którego w zależności od wartości wyrażenia testowego, wykonywane jest określone działanie. Jest instrukcją złożoną, w której możemy dowolnie zagnieżdżać inne instrukcje, w tym kolejne instrukcje `if`. Python pozwala łączyć instrukcje w programie w sposób sekwencyjny (wykonując je jedną po drugiej) i dowolnie zagnieżdżony tak, aby wykonywane były jedynie te spełniające pewne warunki (wybory).

2.1.1. Prosta instrukcja warunkowa

Prosta instrukcja warunkowa przybiera formę testu, po którym wykonywany jest blok instrukcji, o ile zwraca on wynik będący prawdą. Zwracamy szczególną uwagę na znak dwukropka występujący w instrukcji warunkowej po wyrażeniu, którego wartość jest testowana, po którym to występuje również blok będący zestawem instrukcji zapisanych po wcięciu.

Na rysunku 2.1 przedstawiono schemat blokowy prostej instrukcji warunkowej, której ogólna postać wygląda następująco:

```
if wyrażenie:
  blok_instrukcji_T
```



Rysunek 2.1: Instrukcja warunkowa prosta

Na listingu 2.1 przedstawiamy najprostszą postać instrukcji warunkowej z warunkiem zawsze prawdziwym. Jeżeli blok instrukcji warunkowej `if` nie jest złożony, możemy zapisać pojedynczą instrukcję w jednym wierszu za dwukropkiem (i bez wcięcia).

LISTING 2.1: *Warunek zawsze prawdziwy*

```
1 if 1:
2   print("prawda")
3 # OUTPUT: prawda
```

Rozważmy jeszcze jeden przykład przedstawiony na listingu 2.2, w którym testowana jest wartość zmiennej podanej przez użytkownika. Oczekujemy wprowadzenia wartości 0, co jest potwierdzane odpowiednim komunikatem. Natomiast jeśli testowany warunek będzie fałszywy, program przejdzie do instrukcji znajdującej się bezpośrednio za instrukcją warunkową i wykona ją. W szczególnym przypadku może to być kolejna instrukcja `if`.

LISTING 2.2: *Prosta instrukcja* `if`

```
1 a = int(input("Podaj dowolną liczbę całkowitą: "))
2 if a == 0:
3   print("Tak, wprowadziłeś", a)
4 print("Spróbuj kiedyś jeszcze raz!")
```

Stosując `operator morsa` można przesunąć instrukcję z linii 1 listingu 2.2 do warunku instrukcji `if`:

```
if a := int(input("Podaj dowolną liczbę całkowitą: ")) == 0:
```

co pozwoli na wykorzystanie zmiennej `a` w instrukcji `print`.

Na listingu 2.3 prezentujemy kod programu wyświetlającego na ekranie wartość bezwzględną liczby całkowitej podanej przez użytkownika z użyciem prostej instrukcji warunkowej i pomocniczej zmiennej przechowującej wartość wskazywaną przez zmienną `b`. Potrzeba użycia tej zmiennej wynika z faktu, że dokonujemy w bloku instrukcji, gdy liczba jest ujemna, zamiany jej wartości na liczbę przeciwną.

LISTING 2.3: *Obliczanie wartości bezwzględnej*

```
1 c = b = int(input("Podaj dowolną liczbę całkowitą: "))
2 if b < 0:
3   b = -b
4 print("|", c, "| =", b)
```

2.1.2. Instrukcja warunkowa z klauzulą `else`

Teraz rozważymy instrukcję warunkową z klauzulą `else`, która obsługuje przypadek, gdy test wyrażenia w warunku zwraca fałsz. Zauważmy, że po instrukcji `if` oraz instrukcji `else` występuje dwukropek i obowiązkowe wcięcie w bloku instrukcji (wcięcie można pominąć, gdy blok zawiera tylko jedną instrukcję). Zwracamy uwagę, że w obrębie klauzuli `else` nie definiuje się żadnego wyrażenia do testowania.

Ogólna postać rozgałęzionej instrukcji warunkowej jest następująca:

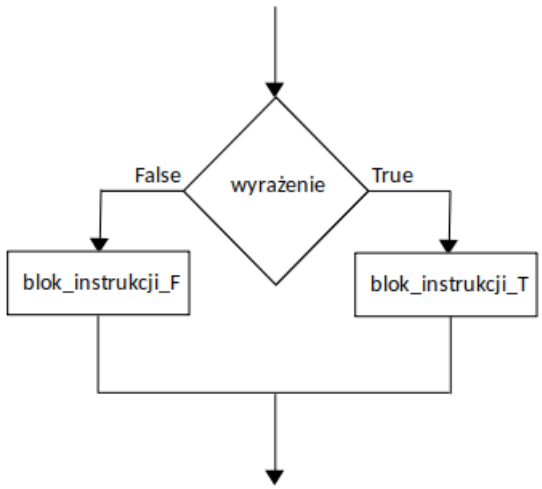
```
if wyrażenie:
  blok_instrukcji_T
else:
  blok_instrukcji_F
```

A jej schemat blokowy przedstawiamy na rysunku 2.2.

Na listingu 2.4 przedstawiamy program, który prosi użytkownika o podanie dwóch liczb zmiennoprzecinkowych, z których pierwsza ma być dzielną, a druga dzielnikiem. Oczywiście niezbędne jest sprawdzenie, czy dzielnik nie jest zerem. Jeśli jest zerem, wypisywany jest na ekranie odpowiedni komunikat i program kończy działanie. Jeśli dzielnik jest różny od zera, wykonywane jest dzielenie, a wynik wyświetlany jest na ekranie i program również kończy działanie.

LISTING 2.4: *Instrukcja warunkowa* `if-else`

```
1 a = float(input("Podaj pierwszą liczbę"))
2 b = float(input("Podaj drugą liczbę"))
3 if b == 0:
4   print("Nie dzielimy przez zero!")
5 else:
6   print(a,":",b,"=",a/b)
```



Rysunek 2.2: Instrukcja warunkowa z klauzulą `else`

Jednym z często spotykanych zadań operatorów logicznych jest zapisywanie w kodzie programu wyrażeń działających tak samo jak instrukcja warunkowa `if`. Rozważmy instrukcję, która w zależności od wartości (prawda lub fałsz) wyrażenia `X`, ustawia zmienną `A` na `Y` lub na `Z`.

```
if X:
  A = Y
else:
  A = Z
```

Możliwe jest zapisanie instrukcji warunkowej, zawierającej pojedyncze instrukcje w blokach, w jednym wyrażeniu (nazywanym często operatorem trójargumentowym lub trójkowym):

```
A = Y if X else Z
```

Wyrażenie to ma dokładnie ten sam efekt co poprzednia czterowierszowa instrukcja `if`, jednak jest łatwiejsze w zapisie i może być wykorzystywane w większych konstrukcjach. Można powiedzieć, że jest to jej skrót. Praktyczne zastosowanie tej konstrukcji pokażemy również w dalszych rozdziałach tego podręcznika.

Na listingu 2.5 przedstawiamy wykorzystanie operatora trójkowego do obliczenia i wyświetlenia na ekranie wartości bezwzględnej liczby całkowitej podanej przez użytkownika.

Listing 2.5: *Zastosowanie operatora trójkowego*

```
1 b = int(input("Podaj dowolną liczbę całkowitą: "))
2 print("|", b, "| = ", b if b >= 0 else -b)
```

2.1.3. Pełna instrukcja warunkowa

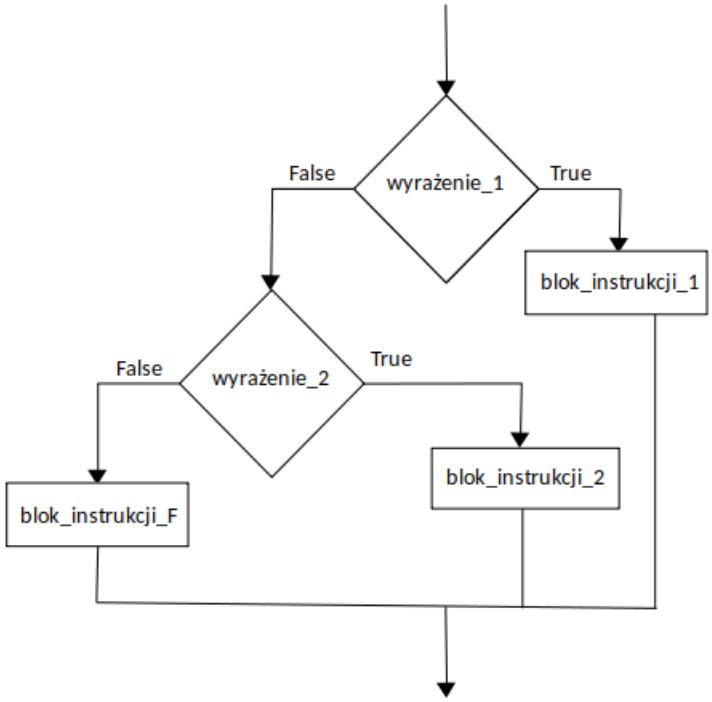
Przedstawimy teraz składnię instrukcji warunkowej `if`, w której występują wszystkie jej klauzule (patrz rysunek 2.3, na którym przedstawiono schemat blokowy pełnej instrukcji warunkowej). Klauzula `elif` daje możliwość rozważenia kolejnego warunku w sytuacji, gdy poprzedni warunek jest fałszywy (jest to skrót od frazy `else if`). Dopiero po sprawdzeniu wszystkich warunków, gdy każdy z nich jest fałszywy, uruchamiany jest blok instrukcji klauzuli `else`. Zauważ, że słowa kluczowe `if`, `elif` oraz `else` są ze sobą powiązane za pomocą wcięć - są wyrównane w pionie. Należy podkreślić, że pełna instrukcja warunkowa może być używana tylko wtedy, gdy warunki występujące po wszystkich klauzulach `elif` wykluczają się nawzajem (nie ma takiej sytuacji, że przynajmniej dwa warunki są spełnione jednocześnie).

```
if wyrażenie_1:
  blok_instrukcji_1
elif wyrażenie_2:
  blok_instrukcji_2
...
else:
  blok_instrukcji_F
```

Rozważmy przykładowy program przedstawiony na listingu 2.6, w którym użytkownik podaje długość boku kwadratu, dla którego następnie wybiera, czy ma być obliczany jego obwód (wymaga podania liczby 2), czy pole (wymaga podania liczby 1). Wprowadzenie liczby różnej od 1 i 2 w trakcie wyboru charakterystyki, która ma być obliczana, spowoduje wypisanie odpowiedniego komunikatu o błędzie na ekranie. Następnie, niezależnie od wyboru, program kończy swoje działanie.

Listing 2.6: *Instrukcja warunkowa* `if-elif-else`

```
1  a = float(input("Podaj długość boku kwadratu:"))
2  if a <= 0:
3    print("Kwadrat nie istnieje")
4  else:
5    wybor = int(input("1. POLE  2. OBWÓD: "))
6    if wybor == 1:
7      print("Pole wynosi: ", a*a)
8    elif wybor == 2:
9      print("Obwód wynosi: ", 4*a)
10   else:
11     print("Zły wybór!")
```



Rysunek 2.3: Schemat blokowy pełnej instrukcji warunkowej

2.2. Instrukcja wyboru

W wersji 3.10 języka Python została wprowadzona instrukcja wyboru `match-case`. Instrukcja ta jako argument przyjmuje wyrażenie i porównuje jego wartość ze zdefiniowanymi schematami. Jeśli wyrażenie jest zgodne ze zdefiniowanym schematem, wykonywane są związane z nim instrukcje. Natomiast jeżeli wartość zmiennej podanej do sprawdzenia nie będzie równa żadnemu ze zdefiniowanych przypadków, program wykona fragment kodu zawarty w ostatnim zdefiniowanym schemacie (często nazywanym domyślnym) oznaczonym znakiem podkreślenia `_`.

Ogólna postać instrukcji `match-case` jest następująca:

```
match wyrażenie:
  case schemat_1:
    blok_instrukcji_1
  case schemat_2:
    blok_instrukcji_2
  ...
  case _:
    blok_instrukcji_przypadku_przeciwnego_do_wszystkich_powyżej
```

Rozważmy przykład zastosowania instrukcji `match-case` przedstawiony na listingu 2.7. Instrukcja ta pobiera obiekt (`rgb`), którego wartość podaje użytkownik, testuje go względem wzorców dopasowania (case 'R', case 'G', case 'B') i wykonuje blok instrukcji, jeśli znajdzie dopasowanie. Jeśli użytkownik poda znak spoza trzech wskazanych lub inny łańcuch znaków, np. łańcuch pusty (`''`) bądź łańcuch zawierający więcej liter, np. `'Green'`, zostanie wykonana instrukcja dla domyślnego przypadku (`case _`) i na ekranie wyświetli się komunikat o złym wyborze.

LISTING 2.7: *Instrukcja wyboru match-case*

```
rgb = input("Wybierz jedną z liter [R|G|B]: ")
match rgb:
  case 'R': print("RED")
  case 'G': print("GREEN")
  case 'B': print("BLUE")
  case _: print("Zły wybór!")
```

Po każdym słowie kluczowym `case` występuje wzorzec dopasowania. Python sprawdza dopasowania, przechodząc przez listę przypadków od góry do dołu. Przy pierwszym dopasowaniu Python wykonuje instrukcje bloku dla tego przypadku, kończy działanie instrukcji `match-case` i kontynuuje dalszą część programu.

Załóżmy, że chcemy napisać program (listing 2.8), który wyświetli komunikat, że wprowadzona przez użytkownika liczba jest cyfrą lub nią nie jest.

LISTING 2.8: *Czy cyfra?*

```
czy_cyfra = int(input("Podaj cyfrę: "))
match czy_cyfra:
  case 0: print('To jest cyfra')
  case 1: print('To jest cyfra')
  case 2: print('To jest cyfra')
  case 3: print('To jest cyfra')
  case 4: print('To jest cyfra')
  case 5: print('To jest cyfra')
  case 6: print('To jest cyfra')
  case 7: print('To jest cyfra')
  case 8: print('To jest cyfra')
  case 9: print('To jest cyfra')
  case _: print("To nie jest cyfra")
```

W instrukcji wyboru operator `|`, oznaczający alternatywę bitową, jest używany do dopasowania dowolnego z wielu wzorców. Jeśli dane wejściowe pasują do któregokolwiek z nich, program drukuje informację, że wprowadzona wartość jest cyfrą. Dzięki temu program z listingu 2.8, możemy zapisać w skróconej postaci, jak na listingu 2.9.

Listing 2.9: *Łączenie warunków*

```
czy_cyfra = int(input("Podaj cyfrę: "))
match czy_cyfra:
  case 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9:
    print('To jest cyfra')
  case _: print("To nie jest cyfra")
```

Instrukcja `match-case` pozwala również na zdefiniowanie zmiennych lokalnych, które będą dopasowywane do argumentu. Do tych zmiennych lokalnych możemy się odwołać w zasięgu pojedynczej klauzuli `case`. Co więcej, możemy doprecyzować, kiedy argument ma się do tych zmiennych dopasować, stosując instrukcję warunkową `if`.

Na listingu 2.10 przedstawiamy program sprawdzający, w którym miejscu na układzie współrzędnych znajduje się punkt o współrzędnych podanych przez użytkownika.

Listing 2.10: *Gdzie leży punkt?*

```
X = float(input("Podaj współrzędną x: "))
Y = float(input("Podaj współrzędną y: "))
punkt = (X, Y)
match punkt:
  case (x, y) if  x > 0 and y > 0:
    print('Punkt (', x, ',',y,') leży w I ćwiartce')
  case (x, y) if  x < 0 and y > 0:
    print('Punkt (', x, ',',y,') leży w II ćwiartce')
  case (x, y) if  x < 0 and y < 0:
    print('Punkt (', x, ',',y,') leży w III ćwiartce')
  case (x, y) if  x > 0 and y < 0:
    print('Punkt (', x, ',',y,') leży w IV ćwiartce')
  case (0, 0):
    print('Punkt leży w początku układu współrzędnych')
  case (x, 0):
    print('Punkt leży na osi OX')
  case (_,_):
    print('Punkt leży na osi OY')
```

Punkt jest krotką (parą) współrzędnych podanych przez użytkownika. Jeżeli punkt leży w pierwszej ćwiartce (`case (x, y)`), to musi być spełniony warunek dopasowania, w którym obydwie współrzędne muszą być dodatnie (`if x > 0 and y > 0`, gdzie zmienne lokalne `x` oraz `y` wiązane są z elementami `X` i `Y` krotki `punkt`, odpowiednio). W przypadku, gdy punkt leży na jednej z osi, np. `OX`, ważne jest spełnienie dopasowania, w którym drugi element krotki musi być zerem (`case (x,0)`). Uwzględniony został również przypadek przeciwny do wszystkich pozostałych (`case (_,_)`) - tu gwarantujący położenie punktu na osi `OY`.

2.3. Instrukcje iteracyjne

Instrukcje iteracyjne, nazywane również pętlami, w każdym języku programowania, a więc również i w Pythonie, służą do realizacji algorytmów z powtórzeniami. W algorytmach takich występują wielokrotne realizacje tych samych działań. Każdy język programowania ma własny zestaw instrukcji iteracyjnych. W języku Python mamy do wyboru dwie takie konstrukcje: `while` oraz `for`.

2.3.1. Pętla `while`

Instrukcja iteracyjna `while` jest najbardziej uniwersalną konstrukcją tego typu w języku Python. W zasadzie wszystkie programy można tworzyć używając jedynie tej instrukcji. Wywołuje powtarzanie wciętego bloku instrukcji dotąd, dopóki wynik testu przeprowadzanego na najwyższym poziomie ma wartość prawdziwą. Jeżeli wynik testu jest fałszywy, to sterowanie przechodzi poza blok instrukcji objęty pętlą `while`. Blok pętli `while` nigdy się nie wykona, jeśli wynik pierwszego testu będzie fałszywy.

Schemat blokowy instrukcji `while` został przedstawiony na rysunku 2.4. W najbardziej złożonej postaci pętla `while` składa się z nagłówka z wyrażeniem testowym, ciała zawierającego blok instrukcji oraz opcjonalnej części `else` wykonywanej, kiedy sterowanie opuszcza pętlę bez napotkania instrukcji `break` przerywającej jej działanie. Instrukcje w bloku są powtarzane dopóki testowane w nagłówku wyrażenie jest prawdziwe.

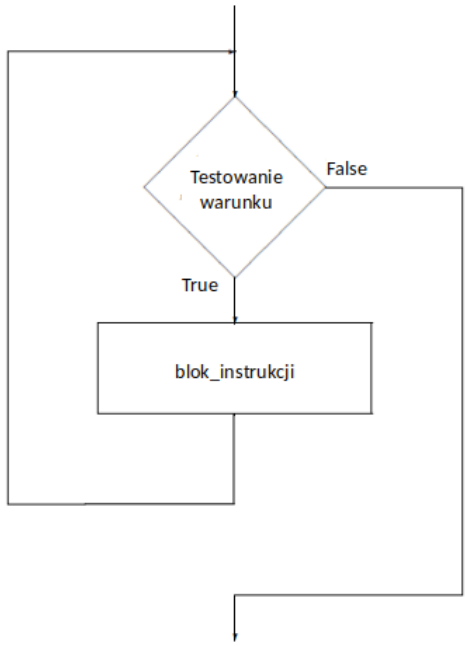
```
while test:          # nagłówek pętli
  blok_instrukcji    # ciało pętli
else:                # klauzula opcjonalna else
  blok_instrukcji    # wykona się, o ile pętli nie zakończy break
```

Najczęściej popełnianymi przez początkujących programistów błędami są źle ustalone warunki kontynuacji pętli:

- warunek kontynuacji pętli nigdy nie jest spełniony - pętla nigdy nie zadziała;
- warunek kontynuacji pętli jest spełniony zawsze - pętla nigdy się nie zakończy.

Takie błędy powodują nie tylko złe działanie programów, ale również, szczególnie w przypadku nieskończonych pętli, niepotrzebne obciążanie zasobów komputera (pamięci RAM) i w efekcie niestabilne działanie systemu operacyjnego.

Na listingu 2.11 przedstawiamy przykład pętli nieskończonej, w której ciele znajduje się instrukcja `pass`. Jest ona elementem zastępczym nie powodującym żadnej akcji i jest wykorzystywana wtedy, gdy składnia wymaga bloku, ale jeszcze nie znamy instrukcji, które by ten blok tworzyły.



Rysunek 2.4: Schemat blokowy pętli `while`

Listing 2.11: *Przykład pętli nieskończonej*

```
1 while 1: pass
```

Zauważ, że ciało pętli zostało umieszczone w jednym wierszu z jej nagłówkiem, po dwukropku. Podobnie do instrukcji warunkowej `if`, taki zapis działa wyłącznie wtedy, gdy ciało nie jest instrukcją złożoną.

I jeszcze jeden przykład (listing 2.12), tym razem pętli, która się nigdy nie wykona (warunek fałszywy).

Listing 2.12: *Przykład pętli, która nigdy się nie wykona*

```
1 while False: print('I tak się nie wykonam!')
```

Python 3.x pozwala również na wpisywanie w kodzie elips (`Ellipsis`), w postaci trzech następujących po sobie kropek (`...`), we wszystkich miejscach, w których może wystąpić wyrażenie. Elipsa, która jest instrukcją „pustą" nie powodującą żadnego działania, jest alternatywą dla instrukcji `pass` lub dla wartości `None`, np. elips możemy użyć do inicjalizacji nazw zmiennych, jeśli nie jest wymagany ich określony typ (`a = ...`).

W przypadku pętli `while` możemy określić z góry liczbę jej powtórzeń, np. pięciokrotne wyświetlenie napisu `PYTHON` (listing 2.13 oraz 2.14) lub liczba powtórzeń może zależeć od spełnienia pewnego warunku, np. wczytujemy liczby z klawiatury dotąd, dopóki użytkownik podaje liczbę niedodatnią (listing 2.15).

Listing 2.13: *Pięciokrotne wyświetlenie na ekranie napisu* ''PYTHON'' *z inkrementacją (zwiększaniem) zmiennej* `i`

```
i = 1            # licznik wyświetleń napisu - warunek startu pętli
while i <= 5:    # pętla kończy się, gdy i > 5
  print("PYTHON")
  i += 1         # zwiększamy licznik o 1
```

Listing 2.14: *Pięciokrotne wyświetlenie na ekranie napisu* ''PYTHON'' *z dekrementacją (zmniejszaniem) zmiennej* `i`

```
i = 5            # licznik wyświetleń napisu - warunek startu pętli
while i > 0:     # pętla kończy się, gdy i=0
  print("PYTHON")
  i -= 1         # zmniejszamy licznik o 1
```

W przypadku programów z listingów 2.13 oraz 2.14, wypiszą one na ekranie pięciokrotnie napis `'PYTHON'`. Różnica tkwi w sposobie odliczania, w pierwszym programie w pętli `while` zmienna będąca licznikiem zwiększa swoją wartość o 1, gdy tymczasem w pętli `while` drugiego programu zmienna ta zmniejsza swoją wartość o 1. W związku z tym, inne są również wartości początkowe dla tej zmiennej oraz inna jest konstrukcja wyrażeń zdefiniowanych w nagłówku pętli.

Na listingu 2.15 przedstawiamy kod, w którym wyrażeniem testowym pętli `while` jest instrukcja `input`, dzięki której użytkownik podawać będzie liczby całkowite, a wprowadzenie przez niego liczby 0 zakończy działanie pętli. Instrukcję `input` umieściliśmy w warunku nagłówka pętli `while`, a podaną przez użytkownika wartość z użyciem operatora morsa przypisaliśmy do zmiennej `liczba`. W ten sposób wykorzystujemy wprowadzane przez użytkownika dane jako wartości testowanego wyrażenia.

Listing 2.15: *Oczekiwanie na podanie liczby całkowitej dodatniej*

```
while (liczba := int(input("Podaj liczbę dodatnią: "))) <= 0:
  print("To nie jest liczba dodatnia. Spróbuj jeszcze raz!")
print("Podałeś liczbę dodatnią:", liczba)
```

Kolejny przykład użycia pętli `while` przedstawiamy na listingu 2.16, gdzie znajduje się kod programu, w którym użytkownik podaje dowolną liczbę całkowitą. Jeśli wprowadzona wartość nie jest dodatnia, to program wypisuje odpowiedni komunikat i kończy działanie. W przeciwnym przypadku, obliczana będzie suma `n` początkowych liczb całkowitych dodatnich. W tym celu wprowadzone zostaną dwie zmienne `i` oraz `suma` (wartością początkową jest zawsze element neutralny działania, w tym przypadku dla dodawania wartość 0), które będą odpowiednio przechowywać liczbę iteracji i sumy cząstkowe.

Listing 2.16: *Obliczanie wartości sumy* `n` *początkowych liczb całkowitych dodatnich*

```
n = int(input("Podaj liczbę całkowitą n: "))
if n > 0:
  i = 1
  suma = 0
  while i <= n:
    suma += i
    i += 1
  if n > 1 and i == 2:
    print(("1 + ... +" if n>2 else '1 +'),n, "= ", end='')
  else:
    print(suma)
else:
  print("Podana liczba nie jest liczbą całkowitą dodatnią!")
```

Z kolei w trakcie wykonania programu z listingu 2.17 użytkownik zostanie poproszony o podanie dowolnej liczby całkowitej. Następnie jeśli wprowadzona wartość nie jest dodatnia, program wypisuje odpowiedni komunikat i kończy działanie. W przeciwnym przypadku, obliczany będzie iloczyn tylko takich `n` początkowych liczb całkowitych dodatnich, które są podzielne przez 3 (reszta z dzielenia przez 3 wynosi 0). W kodzie wprowadzono dwie zmienne `i` oraz `iloczyn` (o wartości początkowej równej 1 - jest to elementem neutralnym mnożenia), które będą odpowiednio przechowywać liczbę iteracji i iloczyny cząstkowe.

Listing 2.17: *Obliczanie iloczynu liczb podzielnych przez 3 nie większych od* `n`

```
n = int(input("Podaj liczbę całkowitą n: "))
if n > 0:
  i = 1
  iloczyn = 1
  while i <= n:
    if i % 3 == 0:
      iloczyn *= i
    i += 1
  print("Iloczyn wynosi:", iloczyn)
else:
  print("Podana liczba nie jest liczbą całkowitą dodatnią!")
```

W instrukcjach iteracyjnych często stosowane są dwie instrukcje pozwalające na sterowanie wykonaniem iteracji: `continue` oraz `break`.

Instrukcja `continue` powoduje natychmiastowe przejście w górę pętli z pominięciem wszystkich instrukcji występujących po niej. Jako przykład rozważmy program z listingu 2.18, który wypisze liczby parzyste mniejsze od liczby całkowitej dodatniej `n` podanej przez

użytkownika. Skonstruujemy pętlę `while` w taki sposób, aby pomijała wszystkie liczby nieparzyste używając w tym celu instrukcji `continue`, która powoduje natychmiastowe przejście do kolejnego wykonania pętli.

LISTING 2.18: *Wyświetlanie liczb parzystych nie większych od* `n`

```
n = int(input("Podaj liczbę całkowitą n: "))
if n > 0:
  i = 0
  while i <= n:
    i = i + 1
    if i % 2 != 0:
      continue
    print(i, end = ', ' if i < n else '')
else:
  print("Podana liczba nie jest liczbą całkowitą dodatnią!")
```

Zwracamy uwagę również na umiejscowienie w kodzie instrukcji zwiększającej wartość zmiennej `i` o 1. Ponieważ zwiększanie musi następować w każdym kroku pętli, nie może ona zostać umieszczona poniżej instrukcji warunkowej `if`, bowiem już przy pierwszym wykonaniu dla wartości `i` równej 1 pętla wykonywałaby się nieskończenie wiele razy.

Z kolei instrukcja `break` powoduje natychmiastowe wyjście z pętli. Rozważmy jako przykład program z listingu 2.19, w którym użytkownik podaje imiona, aby wyświetlić je na ekranie. Po wprowadzeniu słowa `'stop'`, pętla `while` kończy działanie.

LISTING 2.19: *Wyświetlanie wczytanych słów, dopóki użytkownik nie wprowadzi słowa* `'stop'`

```
while True:
  slowo = input('Podaj imię <"stop" kończy wpisywanie>: ')
  if slowo == 'stop':
    break
```

Klauzula `else` pętli `while` wykonywana jest wtedy, gdy w jej ciele nie użyto instrukcji `break`, nawet wtedy, gdy ciało pętli nigdy nie zostanie wykonane. Dzięki temu mamy możliwość wyeliminować stosowanie dodatkowych opcji sprawdzających czy pętla się zakończyła. Na listingu 2.20 przedstawiamy program, który dla podanej przez użytkownika liczby całkowitej, o ile jest ona większa od 1, sprawdza czy jest to liczba pierwsza i wyświetla na ekranie odpowiednie komunikaty.

Instrukcji `continue` i `break` nie należy jednak nadużywać, ponieważ ich bezrefleksyjne użycie może prowadzić do tworzenia skomplikowanego i trudnego do zrozumienia kodu, szczególnie, że w zasadzie można ich użycia uniknąć, zmieniając warunki kontynuacji pętli i/lub nieco inaczej budując kod programu.

Listing 2.20: *Sprawdzanie, czy podana liczba całkowita jest pierwsza*

```
n = int(input("Podaj liczbę całkowitą większą od 1: "))
if n > 1:
 i = n // 2
 while i > 1:
  if n % i == 0:
   print(n,'nie jest pierwsza, bo',i,'jest jej dzielnikiem')
   break
  i -= 1
 else:
  print(n,'jest liczbą pierwszą')
else:
 print("Podana liczba nie jest większa od 1!")
```

2.3.2. Instrukcja iteracyjna `for`

Pętla `for` w Pythonie jest uniwersalnym iteratorem (więcej informacji na temat iteratorów czytelnik odnajdzie w dalszych podrozdziałach tego podręcznika m.in. podrozdziale 8.4), dzięki czemu może przechodzić przez kolejne elementy w takim porządku, w jakim są one umieszczone w danej sekwencji lub innym obiekcie iterowalnym. Instrukcja `for` działa na: łańcuchach znaków, krotkach (np. (1,'Ala',2)), listach (np. [1,2,3,4]), ciągach liczb całkowitych (np. range(1,10), range(1,10,2)) lub słownikach (np. {'wiek': 12, 'płeć':'K'}), innych obiektach wbudowanych, po których można iterować oraz nowych obiektach definiowanych przez użytkownika, które można tworzyć za pomocą klas.

Ogólny format instrukcji iteracyjnej `for` ma postać:

```
for cel in obiekt:      # nagłówek, przypisanie elementów
                        # obiektu do celu
  blok_istrukcji        # ciało pętli, użycie celu
else:                   # opcjonalna klauzula else
  blok_instrukcji_else  # wykona się, gdy nie użyto break
```

Pętla `for` rozpoczyna się od wiersza nagłówka, w którym określa się cel (cele) przypisania wraz z obiektem, który chcemy przechodzić. Po nagłówku znajduje się blok instrukcji, który chcemy powtarzać. Nazwa użyta jako cel w nagłówku pętli jest zazwyczaj (nową) zmienną w zakresie, w którym tworzona jest instrukcja `for`. Może ona być zmieniona w ciele pętli, jednak i tak automatycznie zostanie ustawiona na kolejny element sekwencji w momencie powrotu sterowania do góry pętli. Po wykonaniu się pętli zmienna ta przypisaną ma najczęściej wartość ostatniego elementu sekwencji, o ile nie użyto w ciele pętli instrukcji `break`.

Opcjonalny blok instrukcji w klauzuli `else`, który działa dokładnie tak samo jak w pętli `while`, jest wykonywany tylko wtedy, gdy nie nastąpiło wyjście z pętli w wyniku użycia instrukcji `break`. Oznacza to, że wszystkie elementy sekwencji zostały przetworzone. Omówiona wcześniej dla pętli `while` instrukcja `continue` działa również w podobny sposób w pętli `for`. Zatem pełny format instrukcji `for` może zostać przedstawiony następująco:

```
for cel in obiekt:      # przypisanie elementów obiektu do celu
  blok_istrukcji        # ciało pętli, użycie celu
  if test: break        # wyjście z pętli, pominięcie else
  if test: continue     # powrót na górę pętli
else:
  blok_instrukcji_else  # wykonuje się, gdy nie użyto break
```

Pętla `for` jest pętlą wykorzystującą liczniki, łatwiejszą w zapisie i szybszą w działaniu od pętli `while`. Do generowania indeksów dla pętli `for` można wykorzystać wbudowaną funkcję `range`. W Pythonie 3.x `range` jest iteratorem generującym elementy na żądanie, dlatego aby wyświetlić wszystkie wyniki naraz musimy ją opakować w funkcję `list`. (Więcej informacji o iteratorach czytelnik odnajdzie w podrozdziale 8.4.)

Funkcja `range` jest funkcją tworzącą skończony ciąg arytmetyczny (zawierający liczby całkowite) począwszy od wartości `start`, skończywszy na wartości o jeden mniejszej od `stop` (przedział liczbowy `<start, stop)`). Można ją wywoływać z jednym, dwoma lub trzema argumentami:

```
range(start, stop, krok)
range(start, stop)      # krok = 1
range(stop)             # start = 0, krok = 1
```

Opcjonalny trzeci argument funkcji `range` to `krok`, który, gdy jest użyty, jest dodawany do każdej kolejnej liczby całkowitej. Można zdefiniować postać ogólną dowolnego elementu zakresu w następujący sposób:

- W przypadku gdy `krok > 0`, element j-ty zakresu `r` określony jest wzorem:

  `r[j] = start + krok * j, gdzie j >= 0 i r[j] < stop.`

- W przypadku gdy `krok < 0`, element j-ty zakresu `r` określony jest wzorem:

  `r[j] = start + krok * j, gdzie j >= 0 i r[j] > stop.`

Przykłady zakresów pokazane w trybie interaktywnym Pythona:

- zakres `<0,10)`;

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- zakres <-5,6);

```
>>> list(range(-5, 6))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
```

- co piąta liczba całkowita z zakresu <0,30) zaczynając od 0;

```
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
```

- wszystkie liczby całkowite z zakresu (-10,0> zaczynając od 0 - prawego końca przedziału i zmniejszając każdą kolejną liczbę o 1;

```
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

- funkcja `range` wywołana z jednym zerowym lub ujemnym argumentem sprawia, że w podanym zakresie nie ma liczb;

```
>>> list(range(0))
[]
>>> list(range(-14))
[]
```

- również w zakresie z odwróconymi końcami, nie ma liczb.

```
>>> list(range(1, 0))
[]
```

Rozważmy teraz kilka przykładów wykorzystania funkcji `range` w pętli `for`. Na listingu 2.21 został przedstawiony kod programu, który oblicza i wyświetla na ekranie sumę `n` początkowych liczb całkowitych dodatnich, gdzie `n` jest wartością podawaną przez użytkownika i w przypadku podanej wartości ujemnej lub zera program kończy się, a na ekranie widzimy stosowny komunikat.

LISTING 2.21: *Suma* `n` *początkowych liczb całkowitych dodatnich*

```
1 n = int(input("Podaj liczbę całkowitą: "))
2 if n <= 0:
3   print('Podałeś błędne dane')
4 else:
5   s = 0
6   for j in range(1, n + 1):
7     s += j
8   print("Suma liczb <1,", n, "> =", s)
```

Na listingu 2.22 przedstawiamy program zawierający zagnieżdżone pętle `for`, których zadaniem jest narysowanie na ekranie kwadratu ze znaku gwiazdki `'*'` o wymiarach `n`×`n`, gdzie `n` jest liczbą całkowitą dodatnią podaną przez użytkownika.

LISTING 2.22: *Rysowanie kwadratu o boku* `n`

```
n = int(input("Podaj rozmiar kwadratu: "))
if n > 0:
 for i in range(n):      # wiersze
   for j in range(n):    # kolumny
     print("* ",end="")
   print()
else:
 print('Podałeś błędne dane')
```

Funkcja `range` jest również wykorzystywana w module `random` do generowania wartości losowych z podanego zakresu - jest to funkcja `randrange`. Na listingu 2.23 pokazujemy kilka przykładów dziesięciokrotnego losowania po jednej wartości z różnych zakresów.

LISTING 2.23: *Generowanie liczb losowych*

```
import random
for i in range(20):
   x = random.randrange(20)
   print(x, ' ', end='')
print()
for i in range(20):
   x = random.randrange(1,40)
   print(x, ' ', end='')
print()
for i in range(20, 0 , -1):
   x = random.randrange(1,40,3)
   print(x, ' ', end='')
print()
for i in range(20):
   x = random.randint(1,5)
   print(x, ' ', end='')
```

W ostatnim przykładzie została użyta funkcja `randint`, która w odróżnieniu od funkcji `randrange`, generuje wartości losowe z uwzględnieniem prawego końca przedziału.

2.4. Zadania do samodzielnego rozwiązania

2.4.1. Instrukcja warunkowa

1. Napisz program, który pobierze od użytkownika współczynniki `a` i `b` funkcji liniowej, a następnie obliczy jej miejsce zerowe. Zadbaj o właściwe rozpatrzenie wszystkich przypadków.
2. Napisz program, który pobiera od użytkownika pewną liczbę całkowitą i wypisze na ekranie czy jest parzysta, czy nieparzysta.
3. Napisz program, który prosi użytkownika o podanie uzyskanej oceny w procentach (liczba całkowita) i wyświetli na ekranie słownie odpowiadającą mu standardową ocenę.

```
bardzo dobra       90% - 100%
dobra plus         80% - 89%
dobra              70% - 79%
dostateczna plus   60% - 69%
dostateczna        50% - 59%
niedostateczna     poniżej 50%
```

4. Napisz program, który dla trzech liczb całkowitych `a, b, c` podanych przez użytkownika znajdzie i wypisze na ekranie największą z nich. Spróbuj rozwiązać zadanie z użyciem jak najmniejszej liczby instrukcji warunkowych.
5. Napisz program, który dla danych podanych przez użytkownika zmiennoprzecinkowych liczb `a`, `b` i `c` będących współczynnikami równania $ax^2 + bx + c = 0$ wyznaczy i wypisze na ekranie rzeczywiste rozwiązania tego równania. Rozszerz program o przypadek, w którym rozwiązania są zespolone.
6. Napisz program, w którym użytkownik wczyta promień okręgu do zmiennej `r`, a następnie program obliczy i wyświetli na ekranie pole i obwód koła o promieniu `r`.
7. Napisz program, w którym użytkownik wczyta długości podstaw trapezu oraz jego wysokość, następnie program obliczy i wyświetli na ekranie pole tego trapezu.
8. Napisz program, który korzystając ze wzoru Herona obliczy i wyświetli na ekranie pole trójkąta, dla długości boków podanych przez użytkownika. Czy z dowolnych trzech odcinków zawsze można zbudować trójkąt?
9. Napisz program wypisujący informację na ekranie, w której ćwiartce układu współrzędnych leży podany przez użytkownika punkt. UWAGA: uwzględnij przypadki, gdy punkt leży w środku układu lub na osiach wypisując na ekranie odpowiednie komunikaty.
10. Napisz program, który dla trzech liczb `a`, `b` i `c` wprowadzonych z klawiatury sprawdza, czy są to trójki pitagorejskie. Dodatkowo należy założyć, że `a`, `b` i `c` są dodatnie.

11. Napisz program, który ilustruje działanie operatora logicznego `or` w formacie:

```
a = True
b = True
print("Program ilustruje działanie operatora logicznego or.")
print("True or True -> ", ..., ".", sep = "")
print("False or True -> ", ..., ".", sep = "")
print("True or False -> ", ..., ".", sep = "")
print("False or False -> ", ..., ".", sep = "")
```

    gdzie w miejsce ... wstaw odpowiednie wyrażenie boolowskie zbudowane ze zmiennych `a` i `b` i odpowiednich operatorów boolowskich.
12. Podobnie jak w zadaniu 11, napisz program, który ilustruje działanie operatora logicznego `and`.
13. Korzystając z operatorów `or`, `and` i `not`, napisz program, który dla `p` i `q` podanych przez użytkownika sprawdza pierwsze prawo De Morgana: zaprzeczenie koniunkcji dwóch zdań jest równoważne alternatywie zaprzeczeń tych zdań.
14. Napisz program realizujący rozwiązywanie układu dwóch równań liniowych z dwoma niewiadomymi metodą wyznaczników. Współczynniki wczytane mają być ze standardowego wejścia. W programie uwzględnij wszystkie możliwe rozwiązania.
15. Napisz program symulujący upuszczenie piłki z wieży (bez użycia pętli). Na początek należy zapytać użytkownika o wysokość wieży w metrach. Załóżmy, że grawitacja jest normalna ($g = 9,8\frac{m}{s^2}$) oraz że piłka nie ma prędkości początkowej (piłka nie porusza się, aby wystartować). Niech program wyświetli wysokość piłki nad ziemią po 0, 1, 2, 3, 4 i 5 sekundach, korzystając ze wzoru:

$$odlegosc = \frac{g * x\_sekundy^2}{2}$$

    UWAGA: Piłka nie powinna wchodzić pod ziemię.

    Przykładowe dane wyjściowe:

```
Wpisz wysokość wieży w metrach: 100
W 0 sekundzie piłka jest na wysokości: 100 metrów
W 1 sekundzie piłka jest na wysokości: 95,1 metra
Po 2 sekundach piłka jest na wysokości: 80,4 metra
Po 3 sekundach piłka jest na wysokości: 55,9 metra
Po 4 sekundach piłka jest na wysokości: 21,6 metra
Po 5 sekundach piłka jest na ziemi.
```

16. Napisz program, który prosi użytkownika o podanie pełnego imienia i nazwiska oraz wieku. Jako dane wyjściowe program ma podać użytkownikowi, w którym roku się

urodził. Zabezpiecz program tak, aby nie można było podać ani ujemnego wieku ani wartości większej od 110.
Przykładowe dane wyjściowe:

```
Wpisz swoje imię i nazwisko: Jan Kowalski
Wpisz swój wiek: -32
Wprowadziłeś niepoprawne dane dla wieku. Ponownie
Wpisz swój wiek: 220
Wprowadziłeś niepoprawne dane dla wieku. Ponownie
Wpisz swój wiek: 32
Jan Kowalski urodził się w 1992.
```

17. Napisz program, który pobierze od użytkownika wagę oraz wzrost, a następnie wyznaczy wskaźnik masy ciała ($BMI = \frac{masa[kg]}{wzrost^2[m^2]}$).
18. Napisz program, który będzie spełniał rolę kalkulatora biegowego realizującego następujące obliczenia:
    (a) wyznaczenie czasu ukończenia biegu na danym dystansie na podstawie wczytanego tempa `[min/km]`;
    (b) wyznaczenie tempa biegu na danym dystansie `[min/km]` na podstawie zakładanego czasu ukończenia.

    Zadbaj o to, aby użytkownik miał możliwość wyboru dystansu biegu.
19. Napisz program, który będzie spełniał rolę kalkulatora kwot netto/brutto.
20. Napisz program, który sprawdza czy podany przez użytkownika rok jest przestępny.
21. Napisz program, który dla podanej przez użytkownika aktualnej godziny, sprawdzi jaki kąt tworzą o tej godzinie wskazówki zegara (godzinowa z minutową).
22. Napisz program, który zweryfikuje poprawność cyfry kontrolnej wprowadzonego numeru PESEL.

2.4.2. Instrukcje iteracyjne

1. Do każdego z poniższych punktów napisz odpowiedni program obliczający i wypisujący określoną w tym punkcie wartość:
    (a) sumę wszystkich liczb parzystych od `2` do `100` (włącznie);
    (b) sumę kwadratów wszystkich liczb od `1` do `100` (włącznie);
    (c) sumę potęg liczby 2 dla wykładników od `1` do `63` (włącznie);
    (d) sumę wszystkich liczb nieparzystych pomiędzy `a` i `b` (włącznie), gdzie `a` i `b` są zmiennymi, do których uprzednio należy wczytać dwie liczby całkowite. Dla `a > b` suma powinna wynosić zero.
2. Napisz program, w którym użytkownik wprowadza liczbę naturalną `n`, a następnie program oblicza i wypisuje na ekranie sumę: $1^2 + 2^2 + 3^2 + ... + n^2$.

3. Napisz program, który oblicza sumę wszystkich liczb parzystych od 2 do liczby `n` włącznie (`n` jest liczbą parzystą wprowadzoną przez użytkownika).
4. Napisz program, który rozkłada wprowadzoną przez użytkownika liczbę naturalną na czynniki pierwsze. Na przykład dla liczby `36` (`36 = 2 * 2 * 3 * 3`) algorytm powinien wypisać kolejno ciąg liczb: `2, 2, 3, 3`.
5. Napisz program, który sprawdza czy wprowadzona przez użytkownika liczba jest doskonała (liczba doskonała to taka liczba naturalna, która jest równa sumie wszystkich swoich dzielników, mniejszych od tej liczby, więcej informacji np. `https://www.matemaks.pl/liczby-doskonale.html`.
6. Napisz program, który dla wprowadzonej przez użytkownika liczby naturalnej `n` oblicza jej podwójną silnię (`n!!`). Podwójna silnia jest określona następująco:

   ```
   0!!=1
   1!!=1
   2!!=1*2=2
   3!!=1*3=3
   4!!=1*2*4=8
   5!!=1*3*5=15
   ```

   Wskazówka: Nie ma potrzeby rozważać dwóch przypadków wprowadzanych danych (liczba parzysta, nieparzysta). Można wykonywać mnożenia, korzystając z przemienności, od końca, np.

   ```
   4!!=4*2
   7!!=7*5*3*1
   ```

   czyli rozpoczynamy mnożenie od liczby `n`, kolejno mnożymy przez liczby o 2 mniejsze do momentu, aż `i >= 1`.
7. Napisz program, który dla wprowadzonej przez użytkownika liczby `n` oblicza `n`-ty wyraz ciągu Fibonacciego tj. ciągu określonego rekurencyjnie:

   ```
   a(1)=1
   a(n+2)=a(n+1)+a(n)
   ```

8. Napisz program, który wyświetla listę liczb pierwszych mniejszych od wprowadzonej przez użytkownika liczby naturalnej `n`.
9. Napisz program, który sprawdza czy użytkownik poprawnie wprowadził hasło. Użytkownik może wprowadzać hasło tylko 5 razy. Hasło to: `1979`.
10. Napisz program, który dotąd prosi użytkownika o podanie znaku, aż z podanych przez niego liter można ułożyć słowo `MAMA`. (WAŻNE: ignorujemy wielkość liter.)
11. Napisz program, który wyświetli na ekranie poniższą figurę składającą się z dowolnego znaku oraz liczby znaków w pierwszym wierszu podanych przez użytkownika, np. dla znaku '*' i liczby znaków w pierwszym wierszu równej `4` na ekranie pojawi się:

```
* * * *
* * *
* *
*
* *
* * *
* * * *
```

12. Wiedząc, że $1233 = 12^2 + 33^2$, napisz program, który znajduje wszystkie liczby z przedziału od 1000 do 9999 spełniające taką ciekawą zależność. Program dodatkowo powinien zliczyć, ile jest takich liczb.
13. Napisz program wyświetlający tabliczkę mnożenia dla liczb od 1 do 100 z wykorzystaniem pętli zagnieżdżonej.
14. Napisz program, który znajduje największą i najmniejszą liczbę ze zbioru `n` wylosowanych liczb całkowitych, generowanych losowo z przedziału od 0 do 100 oraz oblicza wartość średnią ze wszystkich wylosowanych liczb. Do losowania liczb użyj, np. funkcji `randint` z biblioteki `random`.
15. Napisz program, który wylosuje dowolną liczbę całkowitą od zera do 10, a następnie prosi użytkownika o jej zgadnięcie tak długo, aż ten poda poprawną wartość. Rozszerz program tak, aby podawał informacje, za którym razem udało się zgadnąć lub o wskazówki typu „*Podana przez ciebie liczba jest większa/mniejsza od wylosowanej*”. Do losowania liczb użyj, np. funkcji `randint` z biblioteki `random`.
16. Napisz program będący modyfikacją poprzedniego programu w taki sposób, aby ograniczyć do 3 prób możliwość zgadywania przez użytkownika wylosowanej liczby.
17. Napisz program, który wypisze pierwsze `n` liczb naturalnych w kolejności rosnącej i malejącej w osobnych kolumnach, jak pokazano poniżej:

```
0 5
1 4
2 3
3 2
4 1
5 0
```

18. Napisz program, który wypisze współrzędne wszystkich punktów kratowych znajdujących się wewnątrz koła o zadanym promieniu.
19. Napisz program, który pobiera od użytkownika liczby do momentu wprowadzenia liczby zero. Zadaniem programu jest wyznaczenie największej z nich oraz średniej arytmetycznej wprowadzanych liczb.
20. Napisz program, który losuje liczbę naturalną z przedziału `[1,1000]`, a następnie wypisuje wszystkie jej dzielniki naturalne.

21. Napisz program, który dla podanej przez użytkownika liczby naturalnej `n` wypisze wszystkie podzielne przez `2` i jednocześnie niepodzielne przez `3` liczby mniejsze lub równe `n`.
22. Napisz program, który dla podanej przez użytkownika dodatniej liczby naturalnej wypisze wszystkie naturalne liczby `n`-cyfrowe.
23. Napisz program, który dla danej naturalnej liczby dodatniej `n` wypisze w porządku malejącym wszystkie parzyste liczby naturalne mniejsze lub równe `n`.
24. Napisz program, który wyświetli na ekranie menu w następującej postaci:

    ```
    0 - wyjście
    1 - wczytaj liczbę
    2 - wyświetl sumę
    3 - czyść pamięć
    ```

    W przypadku wybrania odpowiedniej opcji, program powinien:
    (0) przerwać działanie,
    (1) poprosić użytkownika o podanie kolejnej liczby,
    (2) wyświetlić sumę z wczytanych liczb,
    (3) usunąć dane dotyczące obliczanej sumy.

Rozdział 3

Łańcuchy znaków

Łańcuch znaków (ang. *string*) jest w Pythonie wbudowanym typem danych (reprezentującym uporządkowaną kolekcję znaków) wykorzystywanym do przechowywania i reprezentowania informacji tekstowych oraz opartych na tekstowym zapisie ciągów bajtowych. Z łańcuchami spotkaliśmy się już we wcześniejszych rozdziałach przy okazji omawiania między innymi funkcji `print` czy `input`. W rozdziale tym skupimy się tylko na najpowszechniej stosowanych narzędziach pracujących na łańcuchach znaków i przykładach ich zastosowania. Czytelnika, zainteresowanego poznaniem wszystkich narzędzi służących do pracy z łańcuchami znaków, odsyłamy do kompletnej dokumentacji w podręczniku biblioteki standardowej Pythona.

3.1. Podstawowe informacje

Łańcuchy znaków Pythona zaliczane są do sekwencji niemutowalnych, co oznacza, że ich zawartość jest uporządkowana od lewej do prawej, a samych łańcuchów nie można modyfikować w miejscu.

Rozpoczniemy od omówienia dwóch podstawowych funkcji służących do konwersji pomiędzy pojedynczymi znakami zawartymi w tablicy znaków Unicode a ich reprezentacją (kodem) w postaci liczb naturalnych i na odwrót. Pierwszą z nich jest wbudowana funkcja `chr`, wywołanie której z argumentem `m` zwraca łańcuch znaków reprezentujący znak, którego reprezentacją w Unicode jest liczba całkowita `m`, np. `chr(80)` zwraca łańcuch znaków `'P'`. Zakresem argumentów dla funkcji `chr` jest przedział obustronnie domknięty `<0, 1_114_111>`. Jeżeli wartość argumentu funkcji `chr` jest poza tym zakresem, to zostanie wygenerowany błąd `ValueError`. Na listingu 3.1 przedstawiamy instrukcje, które wyświetlają duże oraz małe litery alfabetu łacińskiego, a także cyfry (w oparciu o funkcję `chr`).

Listing 3.1: *Użycie funkcji* `chr`

```
for j in range(65, 91):       # duże litery alfabetu łacińskiego
  print(chr(j), end = ' ')
print()
for j in range(97, 123):      # małe litery alfabetu łacińskiego
  print(chr(j), end = ' ')
print()
for j in range(48, 58):       # znaki cyfr
  print(chr(j), end = ' ')
print()
```

Z kolei wywołanie funkcji `ord` z argumentem będącym ciągiem reprezentującym jeden znak Unicode zwraca liczbę naturalną reprezentującą kod danego znaku w Unicode, np. `ord('A')` zwraca liczbę `65`. Funkcja `ord` jest funkcją odwrotną do funkcji `chr`, tzn.

```
ord(chr(65)) = 65
chr(ord('A')) = 'A'
```

Na listingu 3.2 przedstawiamy instrukcje wyświetlające kody wszystkich znaków występujących w odpowiednich łańcuchach (zauważmy dość proste w tym przypadku wykorzystanie pętli `for` na sekwencji, którą jest łańcuch znaków). Jest to odwrotne działanie do funkcji `chr` z listingu 3.1.

Listing 3.2: *Użycie funkcji* `ord`

```
for s in "0123456789":
  print(s, ":", ord(s), end = ' ')
print()
for s in "ĄĆĘŁŃÓŚŻŹ":
  print(s, ":", ord(s), end = ' ')
print()
for s in "ąćęłńóśżź":
  print(s, ":", ord(s), end = ' ')
print()
```

Jedną z podstawowych możliwości języka Python jest (oprócz wykonywania obliczeń na liczbach) operowanie na łańcuchach znaków. Łańcuchy mogą być objęte zarówno znakami apostrofu, jak i znakami cudzysłowu. Przedstawiamy poniżej kilka sposobów reprezentowania łańcucha z użyciem interpretera Pythona.

```
>>> 'informatyka'
'informatyka'
>>> "informatyka"
'informatyka'
```

```
>>> 'A\'propos'
"A'propos"
>>> "A'propos"
"A'propos"
```

Interpreter wypisuje łańcuchy znaków w ten sam sposób, w jaki są one wprowadzane. Są one objęte apostrofami lub cudzysłowami i zawierają ewentualnie inne znaki poprzedzone znakiem ukośnika (\) - zwane również sekwencją ucieczki (ang. *escape sequence*), tak aby w sposób dokładny pokazać zawartość napisu, np.

```
\a # alarm, krótki sygnał dźwiękowy
\n # przejście do nowej linii
\' # znak apostrofu
\" # znak cudzysłowia
"Tytuł filmu \"Rejs\""
```

Łańcuch znaków objęty jest parą cudzysłowów, jeśli zawiera tylko apostrofy, w przeciwnym wypadku objęty jest parą apostrofów, np. `"Tytuł filmu 'Rejs'"` lub `'Tytuł filmu "Rejs"'`. Łańcuchy znaków mogą być także objęte potrójnymi znakami apostrofu lub potrójnymi znakami cudzysłowu, co znane jest już czytelnikowi z wcześniejszych wiadomości o komentarzach w Pythonie.

Łańcuchy znaków mogą być **sklejane** (konkatenowane) za pomocą operatora `+` i **powielane** (zwielokratniane) za pomocą operatora `*`. Dzięki temu możemy łączyć łańcuchy w jeden napis, bez względu na znaki ograniczające łańcuchy:

```
>>> print("Ala" + ' ma ' + "kota")
Ala ma kota
```

lub powielać bez konieczności użycia iteracji:

```
>>> print(3 * ":) ")
:) :) :)
```

Oprócz wymienionych wyżej sposobów zapisania łańcuchów znaków w kodzie dostępne są również:

- surowe łańcuchy znaków (ang. *raw strings*) - łańcuch znaków, który tuż przed cudzysłowem lub apostrofem go otwierającym poprzedzony jest (małą lub wielką) literą `r`. Wyłącza on sekwencje ucieczki, co w efekcie powoduje, że Python zachowuje dosłowne znaki ukośników tak, jak zostały wpisane.
  Surowe łańcuchy znaków wykorzystywane są między innymi do reprezentowania ścieżek katalogów w systemach operacyjnych, np. `r"C:\moj_python\zadanie1.py"` czy w wyrażeniach regularnych, np. wzorzec `r'ab{3}'` mówi o dopasowaniu ciągu

znaków zaczynających się od litery `'a'`, za którą bezpośrednio występują dokładnie trzy litery `'b'` - dopasowywanie wzorców obsługiwane jest przez moduł (bibliotekę) `re`;

- literały bajtowe, np. `b'ziel\x01ony'`;
- literały Unicode w wersjach 3.3+ Pythona, np. `u'zielony\u0020czerwony'`.

3.2. Indeksowanie i wykrawanie

Łańcuchy to ciągi znaków, które można indeksować, czyli odwołać się do pojedynczego znaku w napisie używając operatora indeksowania (`[]`) i wpisanego w nim numeru porządkowego tego znaku. Pierwszy znak w łańcuchu ma indeks (numer porządkowy) 0. Nie istnieje osobny typ obejmujący pojedyncze znaki - znak jest po prostu napisem o długości jeden. Łańcuchy znaków w Pythonie nie mogą być modyfikowane, dlatego próba przypisania nowej wartości pod indeksowaną pozycję w łańcuchu powoduje powstanie błędu `TypeError`.

```
>>> s = "programowanie"
>>> s[0] = "P"
Traceback (most recent call last):
File «stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Podłańcuchy znaków mogą zostać wyspecyfikowane za pomocą notacji tzw. *wykrawania* dwóch indeksów przedzielonych dwukropkiem:

```
>>> "programowanie"[0:3]
'pro'
>>> "programowanie"[3:6]
'gra'
>>> "programowanie"[6:]
'mowanie'
```

Indeksy wykrawania posiadają użyteczne argumenty domyślne:

- pominięty pierwszy indeks posiada domyślną wartość zero,
- pominięty drugi indeks domyślnie równy jest długości łańcucha znaków, którego dotyczy wykrawanie.

Operacja wykrawania posiada następującą użyteczną własność: podłańcuchy połączone tym samym numerem porządkowym, z pominięciem w lewym argumencie złączenia

pierwszego indeksu w zakresie wykrawania oraz w prawym argumencie drugiego argumentu zakresu wykrawania, `s[:i] + s[i:]` są równe całemu łańcuchowi `s`, np.

```
>>> "world"[:3] + "world"[3:] == "world"
True
```

Niepoprawne indeksy wykrawania obsługiwane są dość ostrożnie. Indeks, który jest zbyt duży, zastępowany jest długością łańcucha. Z kolei ograniczenie górne, które jest mniejsze od ograniczenia dolnego, powoduje powstanie pustego napisu.

```
>>> "programowanie"[:19]
'programowanie'
>>> "programowanie"[3:2]
''
```

Python umożliwia dwa rodzaje indeksowania - liczbami nieujemnymi oraz ujemnymi, a nawet łączenie obu rodzajów indeksowań, np. łańcuch "kanapka" może być indeksowany liczbami z zakresu od 0 do 6 lub od -7 do -1. Aby wyznaczyć podciąg, licząc od prawej strony danego napisu, można użyć indeksów będących liczbami ujemnymi, np.

```
>>> "programowanie"[-1] # Ostatni znak
'e'
>>> "programowanie"[-2] # Przedostatni znak
'i'
>>> "programowanie"[-3:] # Trzy ostatnie znaki
'nie'
>>> "programowanie"[:-1] # Wszystkie, oprócz ostatniego
'programowani'
```

Bardzo prosto możemy również uzyskać odwrócony napis (pisany wspak). Wystarczy użyć następującej konstrukcji:

```
>>> 'programowanie'[::-1]
'einawomargorp'
```

Ujemne wykrojenia, które przekraczają ograniczenia napisu są skracane,

```
>>> 'programowanie'[-100:]
'programowanie'
```

ale podanie ujemnego indeksu wykraczającego poza zakres łańcucha (nie oznaczającego wykrawania) spowoduje wystąpienie błędu `IndexError`, np.

```
>>> 'programowanie'[-20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Wbudowana w interpreter funkcja `len` zwraca długość łańcucha. Możemy ją wykorzystać do iteracyjnego przejścia po łańcuchu i wypisania każdego znaku, z którego się składa z użyciem pętli `while` (listing 3.3).

LISTING 3.3: *Użycie funkcji* `len`

```
1 slowo = 'Programowanie jest fajne!'
2 i = 0
3 dl = len(slowo)
4 while i < dl:
5   print(slowo[i], end = '_' if i < dl - 1 else '')
6   i += 1
7 print()
```

Dzięki temu, że napisy w Pythonie są obiektami iterowalnymi, możemy do wypisania pojedynczych ich znaków użyć pętli `for` znacznie skracając kod programu (listing 3.4).

LISTING 3.4: *Użycie pętli* `for`

```
1 for i in 'Programowanie jest fajne!':
2   print(i, end = '_' if i != '!' else '')
```

Napisy można porównywać przy użyciu standardowych operatorów relacyjnych: ==, !=, <, <=, > i >=. Wynik porównania zgodny jest z porządkiem leksykograficznym wyznaczonym przez wartości kodów w Unicode znaków biorących udział w porównaniu, np.

```
>>> "Windows" < "linux"
True
>>> "Windows" < "Linux"
False
>>> "mama" < "ala"
False
```

Operator `in` służy do testowania czy dany napis jest podnapisem innego. Z kolei operator `not in` służy do testowania czy dany napis nie jest podnapisem innego. Przykładowo:

```
>>> "gram" in "programowanie"
True
>>> "program" in "programowanie"
True
```

```
>>> "nie" in "programowanie"
True
>>> "grama" in "programowanie"
False
```

3.3. Metody klasy str

W Pythonie łańcuchy (napisy) są obiektami klasy **str**, która ma szereg zdefiniowanych wbudowanych metod operujących na łańcuchach (więcej informacji zainteresowany czytelnik odnajdzie w dokumentacji Pythona https://docs.python.org/library/stdtypes.html#string-methods).

W tym podręczniku nie będziemy przedstawiać wszystkich dostępnych metod klasy **str**, skupimy się tylko na tych najbardziej popularnych.

- **capitalize()** – zwraca kopię napisu z pierwszym znakiem zmienionym na wielką literę, np. wykorzystamy tę funkcję do zmiany pierwszej litery tytułu filmu na dużą. Zrobimy to na dwa sposoby: wypisując zmieniony tytuł na ekran i przypisując zmieniony tytuł do zmiennej tak, aby można było wykorzystać go w dalszej części programu.

```
1 tytul = "czterej pancerni i pies"
2 print('tytul:', tytul)
3 print('capitalize():', tytul.capitalize())
4 zmieniony_tytul =  tytul.capitalize()
5 print('zmieniony_tytul:', zmieniony_tytul)
```

- **count(napis[, poczatek[, koniec]])** – zwraca ilość nienachodzących na siebie wystąpień napisu **napis** w zakresie **[poczatek:koniec]**. Opcjonalne argumenty **początek** i **koniec** są interpretowane tak samo, jak w operacji wykrawania.

```
1 tytul = "czterej pancerni i pies"
2 ile = tytul.count("a")
3 print("count(\"a\"):", ile)
4 ile = tytul.count("er")
5 print('count("er"):',ile)
6 ile = tytul.count("a", 0, 9)
7 print('count("a", 0, 9):',ile)
8 ile = tytul.count("a", 0, 10)
9 print('count("a", 0, 10):',ile)
10 ile = tytul.count("peis")
11 print('count("peis"):',ile)
```

- `endswith(przyrostek[, poczatek[, koniec]])` – zwraca wynik sprawdzenia, czy napis jest zakończony napisem `przyrostek`. Przy wystąpieniu argumentu `początek`, sprawdzenie rozpoczyna się od tego znaku. Przy wystąpieniu argumentu `koniec` porównanie zakończy się na tym znaku.

```
tytul = "czterej pancerni i pies"
print('endswith("pies"):',end='')
print(tytul.endswith("pies"))
print('endswith("pancerni"):',end='')
print(tytul.endswith("pancerni"))
print('endswith("pancerni",0,16 ):',end='')
print(tytul.endswith("pancerni",0,16 ))
```

- `expandtabs([wielkość])` – zwraca kopię napisu ze wszystkimi znakami tabulacji zastąpionymi przez znaki spacji. Jeśli wielkość nie zostanie podana, przyjmuje się rozmiar tabulacji jako 8 znaków.

```
tytul = 'czterej\tpancerni\ti\t\tpies'
print('z tabulacja:',tytul)
print('expandtabs(1):',tytul.expandtabs(1))
```

- `find(podnapis[, początek[, koniec]])` – zwraca najniższy indeks takiego wystąpienia napisu `podnapis`, aby napis był zawarty w wycinku odpowiadającym przedziałowi `<początek:koniec)`. Opcjonalne argumenty `początek` i `koniec` są interpretowane tak samo, jak w operacji wykrawania. Zwraca -1, jeśli napis `podnapis` nie został znaleziony. WAŻNE: Funkcja `find` powinna być używana tylko wtedy, gdy chcemy poznać pozycję napisu `podnapis` w danym napisie. Jeżeli chcemy tylko sprawdzić czy napis `podnapis` występuje w danym napisie, to należy użyć operatora `in`: `podnapis in napis`.

```
tytul = "czterej pancerni i pies"
print('find("pies"):',tytul.find("pies"))
print('find("pies",0, 19):', end='')
print(tytul.find("pies",0, 19))
```

- `isalnum()` – zwraca wynik sprawdzenia, czy wszystkie znaki napisu są znakami alfanumerycznymi i napis składa się przynajmniej z jednego znaku.

```
print('"Asd123klmP".isalnum():',end='')
print("Asd123klmP".isalnum())
print('"Asd12!@3klmP".isalnum():',end='')
print("Asd12!@3klmP".isalnum())
```

- `isalpha()` – zwraca wynik sprawdzenia, czy wszystkie znaki napisu są literami i napis składa się przynajmniej z jednego znaku.

```
print('"A123aaa".isalpha():', end='')
print("A123aaa".isalpha())
print('"Aaaa".isalpha():', "Aaaa".isalpha())
```

- `isdigit()` – zwraca wynik sprawdzenia, czy wszystkie znaki napisu są cyframi.

```
print('"1234".isdigit():',"1234".isdigit())
print('"1234a".isdigit():',"1234a".isdigit())
```

- `islower()` – zwraca wynik sprawdzenia, czy wszystkie litery napisu są małymi literami i napis zawiera przynajmniej jedną mała literę.

```
tytul = "czterej pancerni i pies"
print('tytul[0].islower():',tytul[0].islower())
print('tytul.islower():',tytul.islower())
```

- `isspace()` – zwraca wynik sprawdzenia, czy wszystkie znaki napisu są białymi znakami i napis składa się przynajmniej z jednego znaku.

```
print('" \t\t ".isspace():'," \t\t ".isspace())
print('"".isspace():',"".isspace())
```

- `istitle()` – zwraca wynik sprawdzenia, czy napis ma strukturę tytułu, to znaczy każdy wyraz napisu musi zaczynać się wielką literą i składać wyłącznie z małych liter lub znaków nieliterowych.

```
tytul = 'Czterej pancerni i pies'
print('tytul:', tytul)
print('istitle():', tytul.istitle())
tytul = "Czterej Pancerni I Pies"
print('tytul:', tytul)
print('istitle():', tytul.istitle())
```

- `isupper()` – zwraca wynik sprawdzenia, czy wszystkie litery napisu są wielkimi literami i napis zawiera przynajmniej jedną wielką literę.

```
tytul = "Czterej pancerni i pies"
print('tytul[0].isupper():',tytul[0].isupper())
print('tytul.isupper():', tytul.isupper())
```

- `ljust(szerokość)` – zwraca kopię napisu wyrównaną do lewej w napisie o szerokości `szerokość`. Wypełnienie jest uzyskane za pomocą znaków spacji. Jeśli `szerokość` jest mniejsza od `len(s)` zwracany jest oryginalny napis.

```
napis = 'Ala ma kota'
print('ljust():', napis.ljust(len(napis)+10))
```

- `lower()` – zwraca kopię napisu z wielkimi literami zamienionymi na małe.

```
napis = 'Ala ma kota'
print('lower():',  napis.lower())
```

- `lstrip([chars])` – zwraca kopię napisu z usuniętymi znakami z początku napisu. W przypadku, gdy argument `chars` nie został podany lub ma wartość `None`, usunięte zostaną białe znaki. Jeżeli argument ten jest podany i nie ma wartości `None`, musi być typu napisowego. Z początku napisu, na rzecz którego wywołana została ta metoda, zostaną usunięte znaki wchodzące w skład argumentu `chars`.

```
napis = '\t\tAla ma kota'
print(napis, ': ', end='')
print(napis.lstrip())
print(napis.lstrip("\tAa"))
print(napis.lstrip("\tBCD"))
```

- `partition(sep)` - dzieli łańcuch znaków w miejscu pierwszego wystąpienia separatora `sep` i zwraca trójelementową krotkę zawierającą część przed separatorem, separator oraz część po separatorze. Jeżeli separator nie zostanie znaleziony, zwraca trójelementową krotkę zawierającą cały łańcuch jako pierwszą składową krotki oraz dwa ciągi puste jako pozostałe składowe krotki, np.

```
>>> a = "Ala ma kota"
>>> a.partition(' ')
('Ala', ' ', 'ma kota')
>>> a.partition('b')
('Ala ma kota', '', '')
```

- `replace(stary,nowy[,ile])` – zwraca kopię napisu z wszystkimi wystąpieniami napisu `stary` zastąpionymi przez `nowy`. Jeśli zostanie podany argument `ile`, zostanie zastąpiona tylko podana liczba wystąpień.

```
napis = 'Ala ma kota'
print('replace():', napis.replace('a','x',2))
```

- `rfind(napis[,początek[,koniec]])` – zwraca najwyższy indeks wystąpienia napisu `napis`, takiego aby `napis` był zawarty w przedziale:
  `<początek,koniec)`.
  Opcjonalne argumenty `początek` i `koniec` są interpretowane tak samo, jak w operacji wykrawania. Zwraca -1 jeśli `napis` nie został znaleziony.

```
napis = 'Ala ma kota'
print('rfind():', napis.rfind('a '))
print('rfind():', napis.rfind('a ',0,5))
print('rfind():', napis.rfind('a ',0,2))
```

- `rjust(szerokość)` – zwraca kopię napisu wyrównaną do prawej w napisie o szerokości `szerokość`. Wypełnienie jest uzyskane za pomocą znaków spacji. Jeśli szerokość jest mniejsza od `len(s)` zwracany jest oryginalny napis.

```
napis = 'Ala ma kota'
print('rjust():', napis.rjust(len(napis)+10))
```

- `rstrip([chars])` – zwraca kopię napisu z usuniętymi znakami z końca napisu. W przypadku, gdy argument `chars` nie został podany lub ma wartość `None`, usunięte zostaną białe znaki. Jeżeli argument ten jest podany i nie ma wartości `None`, musi być typu `str`. Z końca napisu, na rzecz którego wywołana została ta metoda, zostaną usunięte znaki wchodzące w skład argumentu `chars`.

```
napis = '\t\tAla ma kota\t\t'
print(napis, ': ', end='')
print(napis.rstrip())
print(napis.rstrip("\tAa"))
print(napis.rstrip("\tBCD"))
```

- `split(sep = None, maxsplit = -1)` - zwraca listę słów w łańcuchu, używając separatora `sep` jako ogranicznika. Jeżeli został podany argument `maxsplit`, wynikowa lista będzie mieć co najwyżej `maxsplit + 1` elementów. Jeżeli argument `maxsplit` nie jest określony lub równy -1, to nie ma limitu na liczbę podziałów. Jeżeli podano argument `sep`, to kolejne jego wystąpienia w łańcuchu nie są grupowane razem i uważa się, że oddzielają one puste łańcuchy. Jeśli argument `sep` nie jest określony lub jego wartość jest równa `None`, to stosowany jest inny algorytm podziału: sąsiadujące białe znaki są traktowane jako jeden separator, a wynik podziału nie będzie zawierać pustych ciągów. W konsekwencji, podział ciągu pustego lub ciągu składającego się tylko z białych znaków da w wyniku listę pustą. Przykładowo:

```
>>> "".split(",")
['']
>>> "".split()
[]
>>> s = "Ala ma kota, który łowi \nmyszy"
>>> print(s)
Ala ma kota, który łowi
myszy
>>> s.split()
['Ala', 'ma', 'kota,', 'który', 'łowi', 'myszy']
>>> s.split(",")
['Ala ma kota', ' który łowi \nmyszy']
```

```
>>> s.split("ma")
['Ala ', ' kota, który łowi \nmyszy']
>>> s.split(maxsplit = 2)
['Ala', 'ma', 'kota, który łowi \n myszy']
```

- `startswith(prefix[, poczatek[, koniec]])` – zwraca wynik sprawdzenia, czy napis zaczyna się napisem `prefix`. Przy wystąpieniu argumentu `poczatek`, sprawdzenie rozpoczyna się od tego znaku. Przy wystąpieniu argumentu `koniec` porównanie zakończy się na tym znaku.

```
napis = '\t\tAla ma kota\t\t'
print('startswith():',napis.startswith("\tA"))
print('startswith():',napis.startswith('\t',0,2))
print('startswith():',napis.startswith("\t\tA"))
```

- `strip([chars])` – zwraca kopię napisu z usuniętymi znakami z początku i końca napisu. W przypadku, gdy argument `chars` nie został podany lub ma wartość `None`, usunięte zostaną białe znaki. Jeśli argument ten jest podany i nie ma wartości `None`, musi być typu napisowego. Z początku i końca napisu, na rzecz którego wywołana została ta metoda, zostaną usunięte znaki wchodzące w skład argumentu `chars`.

```
napis = '\t\tAla ma kota\t\t'
print(napis, end='|\n')
print('strip():', napis.strip())
```

- `swapcase()` – zwraca kopię napisu z małymi literami zamienionymi na wielkie, a wielkimi na małe.

```
napis = 'Ala ma kota'
print(napis)
print('swapcase(): ', napis.swapcase())
```

- `title()` – zwraca kopię napisu zamienioną na strukturę tytułu, to znaczy każdy wyraz napisu zostaje zamieniony na rozpoczynający się wielką literą z pozostałymi literami zamienionymi na małe.

```
napis = 'Ala ma kota'
print(napis)
print('title():', napis.title())
```

- `upper()` – zwraca kopię napisu z wszystkimi literami zamienionymi na wielkie litery.

```
napis = 'Ala ma kota'
print(napis)
print('upper():', napis.upper())
```

- `zfill(szerokość)` – zwraca napis uzupełniony z lewej strony zerami do podanej szerokości. W przypadku, gdy wartość argumentu jest mniejsza od długości napisu, zostanie zwrócony oryginalny napis.

```
print('zfill():', "12345".zfill(20))
```

Do obiektów typu `str`, oprócz wspomnianej już wcześniej wbudowanej funkcji obliczającej i zwracającej długość napisu (`len`), można również użyć funkcji `min` oraz `max` znajdujących i zwracających odpowiednio element minimalny oraz maksymalny łańcucha zgodnie z porządkiem leksykograficznym, np.

```
napis = 'To be, or not to be: that is the question'
print(max(napis), 'na pozycji', napis.find(max(napis)))
print(min(napis), 'na pozycji', napis.find(min(napis)))
```

3.4. Formatowanie łańcuchów

Metoda `format` wykonuje operację formatowania łańcucha znaków. Łańcuch, na rzecz którego metoda ta jest wywoływana, może zawierać pola zastępcze ograniczone nawiasami klamrowymi `{}`. Wszystko, co nie jest zawarte w nawiasach klamrowych, jest traktowane jako tekst dosłowny, który jest kopiowany bez zmian do łańcucha wynikowego. Jeżeli chcemy umieścić znak nawiasu klamrowego w tekście dosłownym, to należy go podwoić: `{{` oraz `}}`. Każde pole zastępcze zawiera indeks numeryczny argumentu pozycyjnego lub nazwę argumentu w postaci słowa kluczowego. Metoda `format` zwraca kopię łańcucha, w którym każde pole zastępcze jest zastępowane przekształconą do łańcucha znaków wartością odpowiadającego mu argumentu. Dla każdego pola zastępczego można określić liczbę pozycji, na której ma być wyświetlana wartość argumentu poprzez podanie jej po dwukropku, np. `{0:3}` oznacza trzy pozycje dla wyświetlanej wartości. Przy czym pole zastępcze `{2:6.2f}` oznacza, że wartość wyświetlana będzie za pomocą liczby typu `float`, dla której możemy podać całkowitą liczbę pozycji, na której ma być ona wyświetlana, w tym liczbę miejsc po przecinku. Dodatkowo w obrębie wyznaczonego pola wyświetlania można określić justowanie: `:<` - do lewej, `:>` - do prawej lub `:^` - wyśrodkowane.

```
s1 = "My name is {0}!".format("Forrest")
print(s1)
name = "Forrest Gump"
age = 18
s2 = "I am {1} and I am {0} years old.".format(age, name)
print(s2)
n1, n2 = 4, 5
s3 = "{0:>3} * {1:<3} = {2:^6.2f}".format(n1, n2, n1 * n2)
print(s3)
```

Jeśli w nawiasach klamrowych nie poda się indeksu numerycznego argumentu pozycyjnego, wtedy Python uzupełni każde pole zastępcze, idąc od lewej, kolejnymi wartościami argumentów.

```
s1 = "My name is {}!".format("Forrest")
print(s1)
n1, n2 = 4, 5
s2 = "{:>3} * {:<3} = {:^6.2f}".format(n1, n2, n1 * n2)
print(s2)
```

Można również wykorzystać w polu zastępczym formatowanie wartości argumentu z konwersją do wartości szesnastkowej `{:x}` lub binarnej `{:b}`.

```
print("The decimal {0} to hex value {0:x}".format(123456))
print("The decimal {0} to binary value {0:b}".format(123456))
```

Podstawowe informacje o metodzie `format` zainteresowany czytelnik odnajdzie w przykładowym rozdziale książki Lutz M., „*Python. Leksykon kieszonkowy*"([4]) znajdującej się pod linkiem `http://pdf.helion.pl/pythl4/pythl4-19.pdf` lub w dokumentacji: `http://docs.python.org/3/library/string.html`.

W Pythonie 3.6 dodano nowy, łatwiejszy sposób formatowania łańcuchów (włączania wartości zmiennych bezpośrednio do ciągów tekstowych) - `f-strings` (PEP498). Aby stworzyć `f-string`, należy poprzedzić łańcuch literą „f". Sam łańcuch można sformatować niemal tak samo, jak w przypadku metody `format()`. Podobnie możemy użyć w łańcuchu pól zastępczych ujętych w nawiasy klamrowe, np.

```
>>> kierunek = 'Informatyka'
>>> f"Kierunek: {kierunek}"
'Kierunek: Informatyka'
```

Możemy również formatować wyświetlany łańcuch określając dynamicznie szerokość pola wyświetlania i precyzję dla liczb typu `float`.

```
>>> total = 45.758
>>> width = 10
>>> precision = 4
>>> f"Your total is {total:{width}.{precision}}"
'Your total is      45.76'
```

W powyższym przykładzie utworzyliśmy trzy zmienne. Pierwsza to pewna liczba rzeczywista, a pozostałe dwie są liczbami całkowitymi. Następnie tworzymy `f-string`, w którym umieszczamy zmienną `total` z formatowaniem: szerokość pola (`width`) ma wynosić 10 znaków, a dokładność liczby rzeczywistej (`precision`) określamy jako 4 cyfry: dwie cyfry części całkowitej plus dwie cyfry po przecinku (wartość jest zaokrąglana). Ta sama zmienna w `f-string`-u może być używana wielokrotnie.

3.5. Zadania do samodzielnego rozwiązania

1. Napisz program, który tworzy i wyświetla na ekranie napis składający się z małych liter alfabetu zawartych w napisie podanym przez użytkownika, ale w odwrotnej kolejności ich występowania, np. dla napisu: `'Ala ma Kota'` program wyświetli: `'atoamal'`.
2. Napisz program, który wyświetli na ekranie tylko duże litery alfabetu znajdujące się w napisie podanym przez użytkownika.
3. Napisz program, który zlicza i wyświetla na ekranie liczbę znaków innych niż litery alfabetu (nie rozróżniamy wielkości liter) znajdujące się w napisie podanym przez użytkownika.
4. Napisz program, który wyświetli na ekranie znak najczęściej występujący w napisie podanym przez użytkownika.
5. Napisz program, który porówna dwa napisy podane przez użytkownika i wyświetli na ekranie komunikat `'Te same napisy'`, jeśli są to te same napisy - `'Napisy różne'` w przeciwnym przypadku.
6. Napisz program, który w oparciu o dwa napisy podane przez użytkownika i wypisuje na ekranie napis składający się z niepowtarzających się znaków występujących w obydwu napisach.
7. Napisz program, który wyświetli na ekranie napis powstały poprzez połączenie dwóch napisów podanych przez użytkownika w taki sposób, aby znalazły się w nim tylko znaki inne niż alfanumeryczne.
8. Napisz program, który wyświetli na ekranie napis powstały poprzez usunięcie z napisu podanego przez użytkownika pierwszego wystąpienia napisu również podanego przez użytkownika. Przykładowo, wynikiem działania programu dla napisów `"abrakadabra"` i `"ab"` będzie napis `"rakadabra"`.
9. Napisz program, który wyświetli na ekranie napis powstały poprzez usunięcie z napisu podanego przez użytkownika, wszystkich wystąpień napisu również podanego przez użytkownika, np. dla napisów `"abrakadabra"` i `"ab"` wynikiem będzie napis `"rakadra"`.
10. Napisz program, który z napisu podanego przez użytkownika tworzy odwrócony napis i wypisuje go na ekranie. Przykładowo, w wyniku działania programu dla napisu `"hello"` na ekranie powinien pojawić się napis `"olleh"`.
11. Napisz program, który wyświetli na ekranie wartość `True`, jeżeli napis podany przez użytkownika jest palindromem, a wartość `False` w przeciwnym przypadku. Przykładowo, w wyniku działania programu dla napisu `"kajak"` na ekranie powinna pojawić się wartość `True`, gdy tymczasem dla napisu `'kalka'` powinna być to wartość `False`.

12. Rozszerz powyższy program tak, aby sprawdzał, czy podane zdanie, po usunięciu wszystkich odstępów i zamianie wielkich liter na małe, jest palindromem. Na przykład dla zdania `"Zakopane na pokaz"` program powinien wypisać na ekranie wartość `True`.
13. Napisz program, który z napisu podanego przez użytkownika tworzy napis będący konkatenacją (czyli złączeniem) tego napisu i jego odwrócenia, następnie wypisuje go na ekran. Przykładowo, w wyniku działania programu dla napisu `"link"` na ekranie powinien pojawić się napis `"linkknil"`.
14. Napisz program, który zaszyfruje podany przez użytkownika napis za pomocą szyfru płotkowego o ustalonej wysokości płotka. Program powinien również umożliwiać odczytanie zaszyfrowanego tekstu. Wyjaśnienie czym jest szyfr płotkowy czytelnik odnajdzie, np. pod linkiem `https://pl.wikipedia.org/wiki/Szyfr_p%C5%82otkowy`.
15. Napisz program, który dla podanych napisów reprezentujących liczby w systemie trójkowym wyświetli ich sumę:
    - z wykorzystaniem systemu dziesiętnego,
    - bez wykorzystania systemu dziesiętnego.
16. Napisz program, który przedstawi podaną przez użytkownika liczbę naturalną w zapisie rzymskim (i odwrotnie).
17. Napisz program, który wyświetli kody `Unicode` wszystkich znaków od `a` do `z`.

Rozdział 4

Funkcje

Funkcje w języku Python używane są w celu stworzenia przejrzystego kodu programu, jego podziału na mniejsze fragmenty, które mogą być użyte dowolną ilość razy w dowolnym czasie. Ogólnie funkcja w programowaniu rozumiana jest w pojęciu szerszym niż w matematyce. W matematyce funkcja zawsze posiada określone argumenty i wartości. W programowaniu zbiór argumentów oraz zbiór wartości funkcji może być zbiorem pustym. Oznacza to, że funkcja może nie pobierać żadnych argumentów, może też nie zwracać żadnych wartości, np. służyć tylko do wyświetlania komunikatu.

W rozdziale tym zostaną przedstawione podstawy tworzenia i używania funkcji, reguły dotyczące zasięgu zmiennych oraz przekazywania argumentów.

4.1. Funkcja - postać ogólna

Instrukcja `def` tworzy obiekt funkcji i przypisuje go do zmiennej o nazwie będącej jednocześnie nazwą funkcji - przypisywane jest odniesienie do obiektu funkcji. Ogólny schemat definicji funkcji jest następujący:

```
def nazwa(arg [[, arg1]...]):   # nagłówek funkcji
  blok_instrukcji               # ciało funkcji
  [return wyrażenie]            # opcjonalna instrukcja
```

Definicja funkcji składa się z sygnatury (nagłówka) funkcji - rozpoczynającego się od słowa kluczowego `def`, następnie występuje nazwa funkcji oraz lista argumentów (czasami nazywanych parametrami). Lista ta może być pusta lub zawierać jeden lub więcej elementów oddzielonych przecinkami. Na końcu nagłówka funkcji zawsze znajduje się dwukropek. Ciało funkcji tworzy pojedyncza instrukcja lub blok instrukcji wykonywanych podczas wywołania funkcji z zachowaniem wcięć o tej samej głębokości na początku każdej linii. Każda funkcja Pythona zwraca pewną wartość. Domyślnie zwracaną wartością jest `None`,

chyba że z wnętrza funkcji zostanie zwrócona inna wartość (za pomocą instrukcji `return`). Instrukcja `return` może pojawić się w dowolnym miejscu ciała funkcji. W momencie jej napotkania, funkcja kończy działanie zwracając wynik do miejsca wywołującego. Jeśli instrukcja `return` nie występuje w kodzie funkcji, kończy ona działanie wraz z ostatnią instrukcją w ciele i sterowanie przepływem programu przechodzi poza jej ciało.

W przeciwieństwie do innych języków programowania (np. `C`), funkcje Pythona nie muszą być w pełni zdefiniowane przed wykonaniem programu. Instrukcje `def` nie są analizowane dopóki nie zostaną wykonane, a kod ciała funkcji nie zostanie wykonany aż do momentu, w którym funkcja ta nie zostanie wywołana. Ponieważ `def` jest instrukcją, może zostać użyta w każdym miejscu, w którym występują instrukcje - wewnątrz instrukcji warunkowej, iteracji, a nawet może być zagnieżdżona w innych instrukcjach `def`.

W Pythonie argumenty przekazywane są do funkcji poprzez przypisanie, czyli referencję do obiektu. Domyślnie obowiązuje przekazywanie pozycyjne, chyba że określimy inaczej. W momencie wywołania funkcji, wartości przekazywane do funkcji domyślnie odpowiadają nazwom argumentów w definicji funkcji od lewej do prawej. W wywołaniu funkcji możemy również przekazywać argumenty w postaci przypisań `nazwa = wartość` lub rozpakowywać dowolną ich liczbę z użyciem wyrażeń `*args` i `**kwargs` (więcej informacji na ten temat czytelnik znajdzie w rozdziale 5).

Zgodnie z dokumentacją PEP8 (`https://peps.python.org/pep-0008/`) zawierającą zalecenia, dzięki którym kod Pythona jest bardziej przejrzysty i czytelny, komentarz dokumentacyjny może, ale nie musi wystąpić bezpośrednio po nagłówku funkcji. Komentarze dokumentacyjne ujmuje się zazwyczaj w potrójne cudzysłowy (`"""`), dzięki czemu mogą one zajmować dowolnie wiele linii.

```
def nazwa(arg [[, arg1]...]):
  """ Opcjonalny komentarz dokumentacyjny jedno
      lub wielowierszowy.
  """
  blok_instrukcji
  [return wyrażenie]
```

Komentarze dokumentacyjne można wpisywać na różne sposoby, ale z zachowaniem kilku wymienionych niżej zasad:

- Potrójny cudzysłów stosuje się nawet wtedy, gdy komentarz zajmuje tylko jeden wiersz (patrz listing 4.1). Jest to dobra praktyka umożliwiająca rozszerzanie komentarza.
- Przed komentarzem i po nim nie pozostawia się pustych wierszy.
- Na końcu komentarza umieszcza się kropkę.

- W przypadku komentarza wielowierszowego opisującego funkcję (patrz listing 4.2):
  - Pierwszy wiersz zawiera krótki opis funkcji (linia 2). Na końcu wiersza jest kropka.
  - Pomiędzy krótkim a pełnym opisem znajduje się pusty wiersz (linia 3).
  - Pełen opis (linie 4-15) zawiera opis znaczenia parametrów (argumentów funkcji) oraz wartości zwracanej przez funkcję. Każdy parametr (linie 10-13) oraz wartość zwracana (linie 14-15) opisane są nazwą oraz typem. W opisie tym odnajdziemy również informacje o obsłudze sytuacji wyjątkowych.

LISTING 4.1: *Komentarz dokumentacyjny jednowierszowy*

```
1 def liczby_pierwsze():
2   """Lista liczb pierwszych z zakresu od 1 do 100."""
```

LISTING 4.2: *Komentarz dokumentacyjny wielowierszowy*

```
1  def podaj_numer_konta(pesel, nazwisko, imie):
2    """Identyfikuje numer konta klienta.
3
4    Funkcja podaj_numer_konta() zwraca numer konta,
5    wykorzystując zadane parametry.
6    W argumencie pesel można umieszczać
7    jedynie numer w Powszechnym Elektronicznym
8    Systemie Ewidencji Ludności (PESEL).
9    Błędny numer PESEL spowoduje zgłoszenie wyjątku.
10   :param pesel: PESEL klienta.
11   :type pesel: str
12   :param nazwisko: Nazwisko klienta.
13   :type nazwisko: str
14   :return: Numer konta klienta.
15   :rtype: str
16   """
```

4.2. Argumenty funkcji

W nagłówku funkcji można definiować dowolną liczbę argumentów (parametrów formalnych), rozdzielonych przecinkami. W Pythonie wyróżniamy cztery sposoby definiowania argumentów funkcji:

- argumenty pozycyjne, wymagane - są to argumenty, które nie mają przypisanej wartości domyślnej, znajdują się na początku listy argumentów;
- argumenty opcjonalne, znajdują się na liście za pozycyjnymi, posiadają wartość domyślną;

- krotka argumentów `*args` - lista (dowolnej długości) nienazwanych argumentów reprezentowana wewnętrznie w funkcji jako krotka (więcej informacji o krotkach można odnaleźć w podrozdziale 5.2);
- słownik argumentów `**kvargs` - słownik (dowolnej długości) nazwanych argumentów, ale niezdefiniowanych w funkcji, muszą się znajdować za argumentami opcjonalnymi (więcej informacji o słownikach można odnaleźć w podrozdziale 5.4).

Jeżeli podajemy argumenty zgodnie z kolejnością ich definiowania w nagłówku, nazwy argumentów można pominąć, natomiast jeżeli zmieniamy ich kolejność lub nie podajemy wszystkich opcjonalnych, należy podawać ich nazwy. Nazwy należy podawać również, jeżeli stosujemy `**kvargs`.

Argumenty przekazywane są do funkcji przez automatyczne przypisanie obiektów do nazw zmiennych lokalnych. W celu zilustrowania działania właściwości przekazywania argumentów rozważmy przykład przedstawiony na listingu 4.3. W momencie wywołania funkcji w linii 7 do zmiennej `arg_f` przypisany zostanie obiekt `-99`, ale `arg_f` istnieje tylko wewnątrz wywołanej funkcji i jego modyfikacja nie ma wpływu na miejsce, w którym funkcja została wywołana - przypisanie wewnątrz funkcji zmienia zmienną lokalną `arg_f` na zupełnie inny obiekt. Potwierdzają to wypisane identyfikatory obiektów zilustrowane na rysunku 4.1.

LISTING 4.3: *Argumenty a współdzielone referencje*

```
1 def funkcja(arg_f):
2   print('id(arg_f) >> przed przypisaniem:', id(arg_f))
3   arg_f = 111
4   print('id(arg_f) >> po przypisaniu:', id(arg_f))
5
6 arg = -99
7 funkcja(arg)
8 print(arg)
9 print('id(arg):',id(arg))
```

```
id(arg_f) >> przed przypisaniem: 139984097122896
id(arg_f) >> po przypisaniu: 139984098168496
-99
id(arg): 139984097122896


------------------
(program exited with code: 0)
Press return to continue
```

Rysunek 4.1: Wynik działania programu z listingu 4.3

Nazwy argumentów mogą początkowo współdzielić przekazywany obiekt, jednak tylko tymczasowo, za pierwszym wywołaniem funkcji. Po ponownym przypisaniu nazwy argumentu w ciele funkcji ten związek zanika.

Natomiast kiedy do funkcji przekazywane są obiekty mutowalne, takie jak listy (podrozdział 5.1) czy słowniki (podrozdział 5.4), modyfikacja takich obiektów w miejscu może pojawić się również po wyjściu z funkcji i przez to mieć wpływ na kod wywołujący. Na listingu 4.4 przedstawiamy kod, w którym lista `L` służy jednocześnie jako wejście i wyjście funkcji, a wynik przypisania do `arg_f[1]` w ciele funkcji ma wpływ na wartości listy `L`, co obrazuje wypisanie jej elementów w funkcji głównej (linia 6).

Listing 4.4: *Argument mutowalny funkcji*

```
1 def funkcja(arg_f):
2   arg_f[1] = 333
3
4 L = [-99, 0, 123, -11]
5 funkcja(L)
6 print(L)
7 # output: [-99,333,123,-11]
```

Należy zachować szczególną ostrożność pracując na argumentach mutowalnych funkcji i pamiętać, że zawsze można taki obiekt w ciele funkcji skopiować, aby wszelkie zmiany dokonywane były tylko lokalnie na kopii we wnętrzu funkcji.

4.3. Wywołanie funkcji

Definicje funkcji nie zmieniają przepływu sterowania programu, czyli kolejności wykonywania instrukcji. Wykonanie programu zaczyna się zawsze od jego pierwszej instrukcji, a potem każda następna instrukcja wykonywana jest pojedynczo, w kolejności od góry do dołu. Instrukcje wewnątrz funkcji nie są wykonywane aż do momentu, gdy funkcja nie zostanie wywołana. Funkcję wywołuje się poprzez jej nazwę oraz aktualne parametry wywołania. Jeżeli funkcja zwraca wartość, to możemy wykorzystać tę wartość w innych instrukcjach, w tym między innymi przypisując ją do zmiennej. Może ona również stanowić argument wywołania innej funkcji czy być wartością wypisywaną na standardowym wyjściu.

W programie w języku Python jedna z funkcji ma szczególne znaczenie. Jest to funkcja o nazwie `main`, która jest funkcją główną programu i odpowiada za sterowanie jego działaniem. Interpreter Pythona przetwarzając pliki źródłowe tworzy między innymi zmienną o nazwie `__name__`. Jeśli plik źródłowy jest plikiem głównym (nie jest wczytywany przez inny skrypt, bowiem wówczas zmienna ta przyjmuje nazwę podstawową skryptu), to zmienna `__name__` przyjmuje wartość `"__main__"`. Zastosowanie instrukcji warunkowej

sprawdzającej jaka jest wartość zmiennej `__name__` zapewnia wykonanie funkcji `main` jedynie w przypadku, gdy skrypt jest właściwym skryptem głównym, dzięki czemu możemy korzystać z niego również w niezmienionej formie w innych skryptach. Na listingu 4.5 przedstawiono szkielet funkcji `main` wraz z jej polecanym wywołaniem.

Listing 4.5: *Wywołanie funkcji main*

```
def main():
  """ Funkcja główna programu. """
  pass
if __name__ == "__main__":
  main()
```

Ponieważ funkcje dzielą program na fragmenty kodu, ważne jest prawidłowe umiejscowienie definicji funkcji i instrukcji je wywołujących. Na listingu 4.6 przedstawiamy niepoprawną kolejność instrukcji ze względu na to, że interpreter Pythona poszukiwać będzie definicji funkcji `komunikat` powyżej wywołania funkcji głównej.

Listing 4.6: *Niepoprawna kolejność instrukcji*

```
def main():
  komunikat()
if __name__ == '__main__':
  main()
def komunikat():
  print("*P*Y*T*H*O*N*")
```

Uruchomienie programu z listingu 4.6 zakończy się zgłoszeniem błędu:

```
Traceback (most recent call last):
  ...
NameError: name 'komunikat' is not defined
```

Poprawna kolejność definiowania funkcji w programie została przedstawiona na listingu 4.7. Natomiast na listingu 4.8 mamy również poprawną, ale i polecaną kolejność definiowanych funkcji tak, aby zawsze u góry ekranu mieć definicję funkcji głównej.

Listing 4.7: *Poprawna kolejność instrukcji*

```
def komunikat():
  print("*P*Y*T*H*O*N*")
def main():
  komunikat()
if __name__ == '__main__':
  main()
```

Listing 4.8: *Poprawna i polecana kolejność instrukcji*

```
def main():
  komunikat()
def komunikat():
  print("*P*Y*T*H*O*N*")
if __name__ == '__main__':
  main()
```

Oprócz argumentów, które mogą być opcjonalne, każda funkcja Pythona zwraca pewną wartość. Domyślnie zwracaną wartością jest `None`, natomiast użycie w ciele funkcji instrukcji `return` pozwala na zwracanie dowolnego wyrażenia:

```
return wyrażenie
```

Wykonanie powyższej instrukcji powoduje obliczenie wartości wyrażenia `wyrażenie`, po czym wartość ta staje się wartością zwrotną danej funkcji, która w tym miejscu kończy swoje działanie. Wartość zwrotna może być pojedynczą wartością, bądź krotką wartości i może zostać zignorowana przez wywołującego funkcję - będzie ona wówczas odrzucona.

Na listingu 4.9 przedstawiamy program obliczający pole kwadratu o długości boku podanej przez użytkownika. Uwzględniony został przypadek, gdy podane przez użytkownika dane nie są poprawne (długość boku zerowa lub ujemna) (linie 3-7). Zwracana przez funkcję wartość odpowiadająca polu kwadratu (linia 4) zostaje przypisana do zmiennej i wykorzystana do wypisania komunikatu na ekran (linia 5). W definicji funkcji `pole` wykorzystana została zmienna `wynik`, której przypisana została wartość wyrażenia obliczającego pole kwadratu (linia 10) i to wartość tej zmiennej jest zwracana przez funkcję (linia 11).

Listing 4.9: *Funkcja obliczająca pole kwadratu*

```
def main():
  b = int(input("Podaj długość boku kwadratu: "))
  if b > 0:
    funkcja = pole(b)  #wywołanie funkcji
    print("Pole kwadratu wynosi ", funkcja)
  else:
    print("Wprowadziłeś błędne dane!")

def pole(a):
  wynik = a*a
  return wynik

if __name__ == '__main__':
  main()
```

Na listingu 4.10 przedstawiamy, w jaki sposób można wykorzystać własność instrukcji `return`, która zanim zwróci wartość, najpierw obliczy ją z wyrażenia, do zmodyfikowania definicji funkcji z listingu 4.9 i skrócenia jej kodu.

Listing 4.10: *Poprawiona funkcja obliczająca pole kwadratu*

```
1  def main():
2    b = int(input("Podaj długość boku kwadratu: "))
3    if b > 0:
4      print("Pole kwadratu wynosi",pole(b))
5    else:
6      print("Wprowadziłeś błędne dane!")
7
8  def pole(a):
9    return a*a
10
11 if __name__ == '__main__':
12   main()
```

Na listingu 4.11 prezentujemy zwracanie przez funkcję więcej niż jednej wartości - krotki wartości (linie 12, 14). Co jest ważne, ponieważ jest to krotka, jej elementy nie muszą być tego samego typu. Program prosi użytkownika najpierw o podanie liczby prób (linia 2), a następnie w każdym przebiegu iteracji użytkownik podaje dwie liczby całkowite (linie 4, 5), które następnie funkcja `min_max` zwraca w porządku rosnącym (linia 6). Zwracana krotka jest rozpakowywana, a jej elementy umieszczane w odpowiednich zmiennych. W ten sposób każdą z tych zmiennych można osobno wykorzystać w dalszej części programu, o ile jest taka potrzeba, bez konieczności odwoływania się do krotki.

Listing 4.11: *Funkcja zwracająca parę liczb w kolejności rosnącej*

```
1  def main():
2    n = int(input("Ile razy chcesz dokonać porównania:"))
3    while n > 0:
4      a = int(input("a = "))
5      b = int(input("b = "))
6      min, max = min_max(a,b)
7      print("(", min, ",",max,")")
8      n = n - 1
9  def min_max(a,b):
10   if a < b:
11     return (a,b)
12   else:
13     return (b,a)
14 main()
```

4.4. Zadania do samodzielnego rozwiązania

1. Napisz funkcję `Wypisz`, która wyświetla na ekranie swój pojedynczy argument. Wywołaj ją przekazując różne typy argumentów: łańcuch znaków, liczbę całkowitą i liczbę zmiennoprzecinkową. Następnie wywołaj ją bez przekazywania argumentu. Co się wtedy dzieje? A co się stanie po przekazaniu dwóch argumentów?
2. Zdefiniuj i przetestuj w programie funkcję `IleCyfr`, która dla dodatniej liczby całkowitej, podanej jako argument funkcji, zwróci liczbę cyfr, z których składa się ta liczba. Następnie w funkcji `main` wywołaj funkcję 10 razy dla wartości argumentu generowanego losowo z przedziału $\langle 1, 50000\rangle$.
3. Zadeklaruj i zdefiniuj funkcję `Odwrotnosc`, której zadaniem jest zwrócenie odwrotności liczby całkowitej podanej jako argument. W funkcji `main` oblicz i wypisz na ekranie sumę odwrotności 10 liczb całkowitych wprowadzonych przez użytkownika, które nie są zerami.
4. Zdefiniuj i wywołaj funkcję `Parzysta`, która zwraca `True` jeśli liczba podana jako argument jest parzysta oraz `False` w przeciwnym przypadku. W funkcji `main` oblicz i wypisz na ekranie, ile było liczb parzystych wśród 100 liczb całkowitych wygenerowanych losowo z przedziału $\langle 1, 100\rangle$.
5. Zdefiniuj funkcję `F1(a,b,c)`, która oblicza i zwraca ile jest w przedziale $\langle a, b\rangle$ liczb, które są wielokrotnościami `c`. W funkcji `main` wczytaj ze standardowego wejścia dwie liczby całkowite `i` i `j`. Jeżeli `j` jest większe od `i`, wczytaj ze standardowego wejścia liczbę całkowitą, której wielokrotności będą zliczane przez funkcję zdefiniowaną `F1` i wypisz zwracaną przez funkcję wartość na ekranie. W przeciwnym przypadku, wypisz komunikat `'Niepoprawne dane'` i zakończ program.
6. Zdefiniuj funkcję `F2(a,b)`, która zwraca liczbę wystąpień w przedziale $\langle a, b\rangle$ takich par liczb `(x,y)`, które spełniają warunek `3x=y`. W funkcji `main` wczytaj ze standardowego wejścia dwie liczby całkowite `i` i `j` . Jeżeli `j` jest większe od `i` , wypisz na ekranie wartość zwracaną przez zdefiniowaną funkcję `F2`. W przeciwnym przypadku wypisz komunikat `'Niepoprawne dane'` i zakończ program.
7. Zdefiniuj funkcję `F3`, która obliczy i zwróci wartość wyrażenia:

$$\frac{a}{a+1} - \frac{a+1}{a+2} - \frac{a+2}{a+4} - \frac{a+3}{a+8} - \ldots - \frac{a+n}{a+2^n}$$

   W funkcji `main`() 10 razy wczytaj ze standardowego wejścia dwie liczby całkowite `a` i `n` i jeżeli są one dodatnie, to wypisz na ekranie wartość zwracaną przez funkcję `F3`. W przeciwnym przypadku wypisz komunikat `'Niepoprawne dane'` i zakończ program.
8. Zdefiniuj funkcję `F4(a,n)`, która obliczy i zwróci wartość wyrażenia:

$$\frac{1}{a} + \frac{1}{a+2} + \frac{1}{a+4} + \frac{1}{a+6} + \ldots + \frac{1}{a+2\cdot n}$$

W funkcji `main` 10 razy wczytaj ze standardowego wejścia dwie liczby całkowite `a` i `n` i jeżeli są one dodatnie, to wypisz na ekranie wartość zwracaną przez funkcję `F4`. W przeciwnym przypadku wypisz komunikat `'Niepoprawne dane'` i zakończ program.

9. Zdefiniuj funkcję `F5(n)`, która wypisze na ekranie wszystkie trójki pitagorejskie (tj. trójki liczb całkowitych `a, b, c` takie, że $a^2 + b^2 = c^2$), składające się z liczb mniejszych od `n`. W funkcji `main` wywołaj zdefiniowaną funkcję `F5` dla dodatniej liczby całkowitej `n` podanej przez użytkownika.
10. Zdefiniuj funkcję o nazwie `rzut_kostka`, która symuluje rzut sześcienną kostką do gry, tzn. funkcja ma zwracać losową liczbę naturalną z przedziału `<1,6>`. Korzystając z tej funkcji, zaprogramuj prostą grę (z dwoma zawodnikami), w której w każdej turze każdy zawodnik rzuca dwoma kostkami (tj. dwukrotnie wywołuje zdefiniowaną funkcję). Turę wygrywa ten, kto wyrzuci więcej oczek. Grę wygrywa ten, kto po `n` turach ma więcej zwycięstw. Jeśli po `n` turach jest remis, gra toczy się do pierwszego zwycięstwa. Komputer ma grać sam ze sobą, a dodatkowo po każdej turze wypisywać na ekranie wylosowaną liczbę oczek oraz aktualny bilans zwycięstw i porażek. Liczbę `n` podaje użytkownik.
11. Napisz program, który oblicza przy pomocy funkcji sumę wyrażenia

$$1 + 2 - 3 + 4 - 5 + ... \pm n$$

dla dowolnego `n`.

12. Napisz program, który oblicza przy pomocy funkcji sumę wyrażenia

$$1 * 2 + 2 * 3 + 3 * 4 + ... + n * (n + 1)$$

dla dowolnego `n`.

13. Napisz funkcję przyjmującą jeden argument - ilość drużyn w lidze piłkarskiej. Zadaniem funkcji jest zwrócenie informacji o tym, ile spotkań należy w tej lidze rozegrać w systemie każdy z każdym (bez spotkań rewanżowych).
14. Zdefiniuj funkcję `F6(a,b,c)` sprawdzającą, czy z odcinków o długościach: `a`, `b`, `c` można zbudować trójkąt.
15. Zdefiniuj funkcję przyjmującą jako argument - prędkość wyrażoną w metrach na sekundę. Zadaniem funkcji jest zwrócenie prędkości wyrażonej w kilometrach na godzinę.
16. Zdefiniuj funkcję sprawdzającą, czy podana jako argument wywołania liczba jest liczbą doskonałą.
17. Zdefiniuj funkcję przyjmującą jeden argument - wynik procentowy uzyskany na kolokwium. Zadaniem funkcji jest zwrócenie oceny odpowiadającej danemu wynikowi.

18. Zdefiniuj funkcję przyjmującą jeden argument - dodatnią liczbę naturalną `n`. Zadaniem funkcji jest obliczenie i zwrócenie sumy `n` początkowych wyrazów ciągu o wzorze ogólnym:
$$a_n = \frac{(-1)^{n+1}}{(n+1)n}$$
.
19. Napisz i przetestuj działanie funkcji `Cezar(napis)`, która szyfruje dany napis Szyfrem Cezara (`https://pl.wikipedia.org/wiki/Szyfr_Cezara`).
20. Napisz funkcję zwracającą ilość spółgłosek występujących w podanym jako argument funkcji napisie.
21. Napisz i przetestuj działanie funkcji przyjmującej jeden argument będący napisem. Zadaniem funkcji jest zwrócenie napisu w taki sposób, aby pierwszy znak, o ile jest małą literą, został zamieniony na literę wielką, np. dla napisu `'wiosna'` funkcja zwróci napis `'Wiosna'`.
22. Napisz i przetestuj działanie funkcji `F7(napis1,napis2)` sprawdzającej czy na odpowiednich pozycjach w obu napisach znajdują się takie same znaki. Funkcja powinna zwrócić liczbę odpowiadającą ilości różnic (lub 0 w przypadku, gdy napisy są identyczne).
23. Napisz funkcję, która na podstawie podanego jako argument napisu, zwróci napis, w którym wszystkie samogłoski znajdujące się na miejscach o indeksach parzystych zostaną zastąpione znakiem `'X'`.
24. Napisz funkcję, która dla napisu podanego jako argument, zwróci najczęściej występującą w nim samogłoskę.
25. Napisz funkcję `F8(napis1,napis2,napis3)`, która zwróci napis będący konkatenacją trzech podanych napisów (przy konkatenacji zastosuj kolejność alfabetyczną).
26. Napisz funkcję `F9(napis1,napis2)` sprawdzającą, czy ze znaków pierwszego napisu można utworzyć drugi napis.

Rozdział 5

Struktury danych

W Pythonie możemy wyróżnić cztery główne struktury danych: listy, krotki, zbiory i słowniki, które różnią się sposobem przechowywania i zarządzania danymi - między innymi podejściem do duplikatów. W tym rozdziale omówimy wszystkie wymienione powyżej struktury danych pod względem ich budowy i zastosowań

5.1. Listy

Najbardziej użyteczną strukturą służącą do grupowania różnych wartości jest `lista`, którą można zapisać jako sekwencję elementów poprzedzielanych przecinkami, umieszczoną w kwadratowych nawiasach.

5.1.1. Listy - podstawowe informacje

Tworząc listę nie musimy martwić się o jednorodność typów jej elementów, gdyż w Pythonie elementy listy nie muszą należeć do tej samej klasy, np.

```
>>> zakupy = ['chleb', 1.95, 'mleko', 2.19, 'masło', 3.20]
```

Elementy listy są indeksowane, tzn. mają przydzielone numery porządkowe zaczynające się od 0 dla elementu pierwszego na liście, a ich indeks rośnie o jeden wraz z każdym kolejnym elementem w strukturze. Indeksy pozwalają na wydobycie z listy wartości znajdującej się w niej pod tym indeksem. Do indeksowania elementów listy można również używać liczb całkowitych ujemnych np. dla listy `a=[2,3,4,5,10]` jej elementy mogą być indeksowane liczbami od `-5` do `-1`.

```
>>> print(zakupy[0], zakupy[2], zakupy[4])
chleb mleko masło
```

Listy mogą być przedmiotem operacji wykrawania i sklejania (konkatenacji), np.

```
>>> zakupy = ['chleb', 1.95, 'mleko', 2.19, 'masło', 3.20]
>>> zakupy[1:-1]
[1.95,'mleko',2.19,'masło']
>>> zakupy[:2] + ['sałata', 2 * 2.2]
['chleb',1.95,'sałata',4.4]
>>> 2 * zakupy[:3] + ['koniec']
['chleb',1.95,'mleko','chleb',1.95,'mleko','koniec']
```

Lista jest strukturą mutowalną, co oznacza że można zmieniać jej poszczególne elementy, np.

```
>>> zakupy = ['chleb', 1.95, 'mleko', 2.19, 'masło', 3.20]
>>> zakupy[1] += 0.23
>>> zakupy
['chleb', 2.18, 'mleko', 2.19, 'masło', 3.20]
>>> lista = ['a', 'b', 6, 12]
>>> lista[0:2] = [1, 2]     # Zastępujemy pewne elementy
>>> lista
[1, 2, 6, 12]
>>> lista[0:2] = []         # Usuwamy pewne elementy
>>> lista
[6, 12]
>>> lista[1:1] = ['c', 'd'] # Wstawiamy elementy do środka listy
>>> lista
[6, 'c', 'd ', 12]
>>> lista[:0] = lista       # Wstawiamy listę na początek
>>> lista                   # (w tym przypadku powielamy jej elementy)
[6, 'c', 'd ', 12, 6, 'c', 'd', 12]
>>> lista[:] = []           # Usuwamy wszystkie elementy listy
>>> lista
[]
```

Na listingu 5.1 przedstawiamy przykładowy kod programu, który tworzy listę kwadratów liczb naturalnych od 1 do 20, a potem wyświetla je na ekranie w odwrotnej kolejności. Dodawanie elementów do początkowo pustej listy odbywa się za pomocą instrukcji `append`, która jest metodą klasy `list`. Wyświetlane elementy na ekranie poprzedzone są indeksem, pod którym ta wartość znajduje się w liście. W tym celu stosujemy funkcję `enumerate`, która dodaje licznik do każdego elementu iterowalnego obiektu i zwraca obiekt `enumerate`, który jest krotką `(indeks, wartość)`, jako ciąg wyjściowy.

Listing 5.1: *Użycie metody* `append`

```
a = list()                          #deklaracja pustej listy
for i in range (20):
  a.append((i+1)**2)              #dodawanie kolejnych elementów
print(a)                            #wyświetlenie listy
for i, v in enumerate(a):
  print(i,':',v,", ",end="") #wyświetlenie par (indeks,element)
print()
```

Listy można łączyć ze sobą używając operatora sklejania (+), dlatego dodawana do listy wartość musi być ujęta w nawiasy kwadratowe, co pokazano na listingu 5.2. Można też użyć pewnego skrótu `(a+=i+1,)`, ponieważ postawienie przecinka już zmusza interpreter Pythona do traktowania pojedynczego elementu jako jednoelementowej struktury sekwencyjnej (np. listy). Do wyświetlania elementów listy ponownie użyjemy funkcji `enumerate`, tym razem z drugim opcjonalnym argumentem określającym wartość początkową dla licznika.

Listing 5.2: *Łączenie list*

```
a = list()                  #deklaracja pustej listy
for i in range (20):
  a += [(i+1)**2]         #dodawanie kolejnych elementów
  #a += (i+1)**2,         #alternatywny sposób dodawania elementów
for i, v in enumerate(a,1):
  print(i,':',v)
print()
```

Na listingu 5.3 przedstawiamy program, który wypełnia 10-cio elementową listę liczbami rzeczywistymi wprowadzonymi przez użytkownika i oblicza ich sumę.

Listing 5.3: *Suma elementów listy*

```
a = list()
for i in range (10):
  f = float(input("Podaj element listy:"))
  a += f,
print(a)
suma = 0
for i in a:
  suma += i
print("Suma elementów wynosi:", suma)
```

Na listingu 5.4 przedstawiamy zmodyfikowany program obliczania sumy elementów listy wykonujący sumowanie w tej samej pętli, w której z liczb podawanych przez użytkownika tworzona jest lista.

Listing 5.4: *Suma elementów listy z użyciem jednej pętli*

```
a = list()
suma = 0
for i in range(10):
  a.append(float(input("Podaj "+str(i + 1)+" element:")))
  suma += a[i]
print("Twoja lista:\n", a)
print("Suma elementów wynosi:", suma)
```

Na listingu 5.5 przedstawiamy z kolei program, który wypełnia 10-elementową listę całkowitymi liczbami losowymi z przedziału <1,10> i oblicza iloczyn elementów listy.

Listing 5.5: *Iloczyn elementów listy*

```
from random import randint
a = list()
for i in range (10):
  a += randint(1,10),
iloczyn = 1
print("Twoja lista: ", a)
for i in range(10):
  iloczyn *= a[i]
print("Iloczyn wynosi: ", iloczyn)
```

Na listingu 5.6 prezentujemy zmodyfikowany program obliczający iloczyn elementów listy w tej samej pętli, w której lista jest generowana losowo.

Listing 5.6: *Iloczyn elementów listy z użyciem jednej pętli*

```
from random import randint
a = list()
iloczyn = 1
for i in range(10):
  a.append(randint(1,10))
  iloczyn *= a[i]
print("Twoja lista: ", a)
print("Iloczyn elementów wynosi:", iloczyn)
```

Do list mogą być stosowane wbudowane funkcje: **len**, **min**, **max** i **sum**.

```
>>> lista = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
>>> len(lista)
12
>>> min(lista)
28
```

```
>>> max(lista)
31
>>> sum(lista)
365
>>>
```

Możliwe jest **zagnieżdżanie** list, tzn. tworzenie list, których elementami są inne listy.

```
>>> b = [2, 3]
>>> a = [1, b, 4]
>>> a
[1, [2, 3], 4]
>>> len(a)
3
>>> len(a[1])
2
>>> a[1]
[2, 3]
>>> a[1][0]
2
```

Można usunąć element listy o podanym indeksie, jak również wycinek listy, przy pomocy instrukcji `del`.

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

5.1.2. Wybrane metody klasy list

- `append(value)` - dodaje obiekt `value` na końcu listy.

  ```
  >>> a = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
  >>> a.append([4, 4, 4])
  >>> a[0].append("Python")
  ```

```
>>> a
[[1, 1, 1, 'Python'], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
```

- `extend(iterable)` - dodaje elementy obiektu iterowalnego `iterable` do listy.

```
>>> a = [3, 6, 8]
>>> a.extend([3, 7, 6])
>>> a
[3, 6, 8, 3, 7, 6]
```

- `insert(index, value)` - wstawia obiekt `value` do listy w miejscu określonym przez `index`.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.insert(3, 123)
>>> a
[3, 6, 8, 123, 3, 7, 6]
```

- `remove(value)` - usuwa pierwsze wystąpienie obiektu `value` z listy. Jeżeli obiekt `value` nie występuje w liście, to wyrzuca wyjątek `ValueError`.

```
>>> a = [3, 6, 8, 123, 3, 7]
>>> a.remove(3)
>>> a
[6, 8, 123, 3, 7]
```

- `clear()` - usuwa wszystkie elementy z listy.

```
>>> a = [3, 6, 8, 123, 3, 7]
>>> a.clear()
>>> a
[]
```

- `count(value)` - zwraca liczbę wystąpień obiektu `value` w liście.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.count(6)
2
>>> a.count(0)
0
```

- `pop(index=-1)` - usuwa element z podanej pozycji listy (domyślnie ostatni) oraz zwraca jego wartość. Wyrzuca wyjątek `IndexError`, jeżeli lista jest pusta lub indeks jest poza zakresem.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.pop()
6
>>> a
[3, 6, 8, 3, 7]
>>> a.pop(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

- `index(value, start=0, stop=9223372036854775807)` - zwraca indeks pierwszego wystąpienia obiektu `value` w liście, opcjonalnie w zakresie `<start,stop)`. Jeżeli obiekt `value` nie występuje w liście, to wyrzuca wyjątek `ValueError`.

```
# indeks elementu o wartości 7
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.index(7)
4
# indeks elementu o wartości 8 w zakresie <3,5)
>>> a.index(8,3,5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 8 is not in list
```

- `reverse()` - odwraca kolejność elementów na liście.

```
>>> a = [123, 8, 7, 6, 3]
>>> a.reverse()
>>> a
[3, 6, 7, 8, 123]
```

- `sort(key=None, reverse=False)` - sortuje listę w porządku niemalejącym i zwraca `None`. Sortowanie jest stabilne, tzn. zachowana jest kolejność równych elementów. Jeżeli podano funkcję `key`, to stosuje się ją do każdego elementu listy i sortuje zgodnie z wartościami funkcji `key`. Flaga `reverse` może być ustawiona na sortowanie w porządku nierosnącym.

```
>>> a = [6, 8, 123, 3, 7]
>>> a.sort()
>>> a
[3, 6, 7, 8, 123]
>>> a.sort(reverse = True)
```

```
>>> a
[123, 8, 7, 6, 3]
>>> a = [3, 7, 2, -3, 9, 7, -4, -8, -5, 8]
>>> a.sort()
>>> a
[-8, -5, -4, -3, 2, 3, 7, 7, 8, 9]
>>> a.sort(reverse=True)
>>> a
[9, 8, 7, 7, 3, 2, -3, -4, -5, -8]
>>> a.sort(key=abs)
>>> a
[2, 3, -3, -4, -5, 7, 7, 8, -8, 9]
>>> a.sort(key=abs, reverse=True)
>>> a
[9, 8, -8, 7, 7, -5, -4, 3, -3, 2]
>>> a = ['Adam','Ela','Marian','Władysław']
>>> def F(L):
...     return len(L)
...
>>> a.sort(key = F)
>>> a
['Ela','Adam','Marian','Władysław']
>>> a.sort(key = F, reverse = True)
>>> a
['Władysław','Marian','Adam','Ela']
>>> a.sort(key = len)
>>> a
['Ela','Adam','Marian','Władysław']
```

- `copy()` - tworzy płytką kopię listy. Płytka kopia tworzy nową listę, a następnie (w miarę możliwości) wstawia do niej referencje do obiektów znalezionych w oryginale.

```
>>> a = [3, 6, 8]
>>> a.extend([3, 7, 6])
>>> b = a.copy()
>>> b
[3, 6, 8, 3, 7, 6]
>>> id(a) == id(b)
False
```

```
>>> old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
>>> new_list = old_list.copy()
>>> old_list.append([4, 4, 4])
>>> old_list
[[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
>>> new_list
[[1, 1, 1], [2, 2, 2], [3, 3, 3]]
>>> old_list[0].append("Python")

""" Zmiana elementu o indeksie 0 w old_list
    jest widoczna także w new_list """

>>> old_list
[[1, 1, 1, 'Python'], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
>>> new_list
[[1, 1, 1, 'Python'], [2, 2, 2], [3, 3, 3]]

""" Bowiem new_list[0] jest referencją do old_list[0] """

>>> id(old_list[0]) == id(new_list[0])
True
```

Na listingu 5.7 przedstawiamy różnicę w czasie tworzenia listy trzema różnymi operacjami: rozszerzaniem `extend`, kopiowaniem `copy` i wykrawaniem `[:]`. Do obliczenia czasu działania każdej z nich wykorzystujemy funkcję `time` z modułu `time`.

LISTING 5.7: *Przykład zastosowania metody* `copy`

```
1  from time import time
2  old_list = 10**7 * [2, 3, 5, 7, 11]
3  t = time()
4  new_list = [].extend(old_list)
5  print("list.extend: ", time() - t)
6  t = time()
7  new_list = old_list.copy()
8  print("list.copy: ", time() - t)
9  t = time()
10 new_list = old_list[:]
11 print("list slicing [:]:", time() - t)
```

Wynik wykonania programu potwierdza, że metoda `copy` jest najszybsza w swoim działaniu:

```
list.extend:  0.8033039569854736
list.copy:  0.6313166618347168
list slicing [:]: 0.8029310703277588
```

Na listingu 5.8 przedstawiamy tworzenie listy z listy już istniejącej, co w efekcie prowadzi do utworzenia listy `B` będącej referencją do obiektu `A`.

LISTING 5.8: *Przykład działania na listach dwuwymiarowych*

```
A = [0,1]
B = []
for i in range(2):
  B.append(A)
print('A[0]:',id(A[0]))
print('B[0][0]:',id(B[0][0]))
print('B[1][0]:',id(B[1][0]))
print(B)
A[0]=100
print(B)
```

Potwierdza to wynik działania programu, w którym wypisywane są `id` zerowego elementu listy `A` oraz zerowych elementów wierszy macierzy `B`. Dodatkowo ponieważ macierz `B` powstała jako lista dwóch referencji do obiektu `A`, każda modyfikacja elementów obiektu `A` jest widoczna również w macierzy `B`:

```
A[0]: 140643489186000
B[0][0]: 140643489186000
B[1][0]: 140643489186000
[[0, 1], [0, 1]]
[[100, 1], [100, 1]]
```

5.1.3. Argumenty wywołania programu

Wbudowany moduł `sys` zawiera polecenia związane z Pythonem i jego środowiskiem. Pozwala na obsługę argumentów wywołania programu z wiersza poleceń, dzięki czemu można tworzyć programy jeszcze bardziej elastycznie dostosowane do potrzeb użytkownika. Zmienna `sys.argv` jest listą łańcuchów znaków będących argumentami, z którymi program został wywołany z wiersza poleceń. Przy czym, pierwszy element `argv[0]` jest nazwą skryptu Pythona, w zależności od systemu operacyjnego, może być to pełna ścieżka lub nie. Jeżeli polecenie zostało wykonane przy użyciu opcji wiersza poleceń -c interpretera, to `argv[0]` jest ustawione na ciąg `'-c'`. Jeżeli żadna nazwa skryptu nie została przekazana do interpretera Pythona, to `argv[0]` jest pustym ciągiem znaków.

Listing 5.9: *Wypisanie argumentów wywołania programu jako listy*

```
import sys
def main():
  print(sys.argv)
if __name__ == "__main__":
  main()
```

W wyniku wywołania programu z listingu 5.9 na ekran wypisana zostanie zawartość `sys.argv`.

```
~$ python3 main_arg_1.py 1 2 3 "Ala" [1,2,3]
['main_arg_1.py', '1', '2', '3', 'Ala', '[1,2,3]']
```

Innym sposobem wypisania argumentów wiersza poleceń, zilustrowanym na listingu 5.10, jest użycie pętli `for` i odrębne wypisanie każdego elementu listy `sys.argv`.

Listing 5.10: *Wypisanie argumentów wywołania programu w osobnych liniach*

```
import sys
def main():
  for a in sys.argv:
    print(a)
if __name__ == "__main__":
  main()
```

W wyniku wywołania programu z listingu 5.10 na ekran wypisana zostanie zawartość `sys.argv`.

```
~$ python3 main_arg_2.py 1 "Ala" [1,2,3]
main_arg_2.py
1
Ala
[1,2,3]
```

Jest jeszcze trzeci sposób wypisania listy argumentów wywołania programu z listingu 5.11 z użyciem pętli `for`, ale również z wykorzystaniem funkcji `range` i indeksowania.

Listing 5.11: *Wypisanie argumentów wywołania programu z użyciem funkcji* ***`range`***

```
from sys import argv
def main():
  for j in range(len(argv)):
    print(argv[j])
if __name__ == "__main__":
  main()
```

5.2. Krotki

Krotka (ang. *tuple*) to sekwencja, której zawartości nie można zmienić - sekwencja niemutowalna (niemodyfikowalna). Podczas tworzenia krotki używany jest nawias okrągły, np.:

```
krotka = (1, 2, 3, 4) # Czteroelementowa krotka
```

Wartości krotki rozdzielone są przecinkami. Pusta krotka ma postać: `krotka = ()` natomiast krotka z jednym elementem ma postać: `krotka = (1,)` (lub `krotka = 1,`). Zwracamy uwagę, że przecinek w krotce jednoelementowej jest obowiązkowy. Bez przecinka nie jest to obiekt krotki, tylko liczba całkowita.

Ponieważ krotka jest sekwencją niemodyfikowalną, „obsługuje" ona wszystkie te operacje, które nie powodują zmiany jej zawartości, czyli:

- indeksowanie (jedynie do pobierania wartości elementów);
- metody takie jak `index()`;
- funkcje wbudowane, takie jak `len()` , `min()` i `max()`;
- wykrawanie;
- operator `in`;
- operatory sklejania (konkatenacji) `+` i powielania `*`.

Krotka jest obiektem iterowalnym, można zatem zastosować pętlę `for` do przechodzenia przez jej elementy. Dla przykładu wypiszemy imiona znajdujące się w krotce o nazwie `imiona` wykorzystując pętlę `for` i indeksowanie.

```
1 imiona = ('Jakub','Bartosz','Max','Filip',)
2 for n in range(len(imiona)):
3   print("imiona[", n, "] = ",imiona[n],".",sep = "")
```

Krotki można ze sobą łączyć (konkatenować) oraz je powielać (sklejać), np.

```
1 krotka1 = (1, 2, 3, 4,)
2 krotka2 = (10, 20, 30, 40,)
3 print("Krotka1 = ", krotka1, ".", sep = "")
4 print("Krotka2 = ", krotka2, ".", sep = "")
5 print()
6 krotka = krotka1 + 3*krotka2
7 print("krotka1 + 3*krotka2 = ",krotka, ".",sep = "")
```

Na listingu 5.12 przedstawiamy kod programu, który używając wbudowanych funkcji `min` oraz `max` szuka i wypisuje wartości elementu najmniejszego i największego krotki oraz indeksy, pod którymi się znajdują.

Listing 5.12: *Szukanie elementów* `min` *i* `max` *w krotce*

```
krotka1 = (11, 2, 332, 4)
krotka2 = (1, 20, 30, 40)
print("Krotka1 = ", krotka1, ".", sep = "")
print("Krotka2 = ", krotka2, ".", sep = "")
print()

krotka = krotka1 + krotka2
print("Krotka = ", krotka, ".", sep="")

mn = min(krotka)
mx = max(krotka)
print("min:krotka[",krotka.index(mn),"]=",mn,".",sep="")
print("max:krotka[",krotka.index(mx),"]=",mx,".",sep="")
```

Krotki są znacznie szybsze podczas obliczeń niż listy. Używamy ich wtedy, kiedy mamy do czynienia z elementami, które nie powinny być podatne na modyfikacje, jak np. spis dni tygodnia lub daty w kalendarzu. Krotki zapewniają bezpieczeństwo, że żaden z ich elementów, jak i one same, nie zmienią się podczas przeprowadzanych operacji. Używamy ich do przechowywania różnych typów danych (ang. *heterogeneous data types*). Mówimy wtedy, że mamy do czynienia z typami heterogenicznymi. Krotki można zagnieżdżać, a odwoływanie się do pojedynczego elementu w zagnieżdżonej krotce wymaga użycia dodatkowego operatora indeksowania, np. na listingu 5.13 zdefiniowano krotkę, w skład której wchodzą dwie zagnieżdżone krotki. W odwołaniu `Tuple[2]` identyfikowany jest trzeci element krotki, ten o indeksie 2, który również jest krotką. Zatem, aby pokazać jej ostatni element, którym jest napis `Cześć`, musimy ponownie użyć operatora indeksowania i numeru pozycji, na której ten napis się znajduje, czyli `Tuple[2][2]`.

Listing 5.13: *Zagnieżdżanie krotek*

```
Tuple = ("Python", (2, 4, 6), (4, 5.6, "Cześć"))

print("Zawartość krotki:", Tuple)
print("Elementy krotki Tuple:")
print("\t* Pierwszy:", Tuple[0])
print("\t* Drugi:", Tuple[1])
print("\t* Trzeci:", Tuple[2])
print("\t* Ostatni:", Tuple[-1])
print("\t* Dwa pierwsze:", Tuple[:-1])
print("\t* Pierwszy w ostatnim elemencie:", Tuple[-1][0])
print("\t* Ostatni w ostatnim elemencie:", Tuple[-1][-1])
print("\t* Dwa ostatnie w drugim elemencie:", Tuple[1][1:])
```

Jednym z zastosowań krotki jest używanie jej jako opcjonalnego argumentu specjalnego funkcji oznaczonego `*args`. Forma `*args` określa, że wszelkie dodatkowe nienazwane argumenty pozycyjne wywołania są gromadzone w krotce i są związane z nazwą `args`. Dzięki temu specjalnemu argumentowi, możemy tworzyć funkcje o dowolnej liczbie argumentów, w tym funkcję bezargumentową.

Na listingu 5.14 przedstawiamy praktyczne zastosowanie specjalnego argumentu, który gromadzi w krotce `args` liczby całkowite przekazywane jako argumenty, aby obliczyć i zwrócić ich sumę. W przypadku wywołania funkcji bez argumentów, funkcja zwraca 0.

LISTING 5.14: *Dowolna liczba nienazwanych argumentów pozycyjnych*

```
1  def main():
2    print(func())
3    print(func(3))
4    print(func(3, 5))
5    print(func(31, 28, 31))
6    print(func(3, 5, 31, 28))
7    print(func(3, 5, 8, 31, 28))
8  def func(*args):
9    suma = 0
10   if len(args) != 0:
11     suma = sum(args)
12   return suma
13 if __name__=='__main__':
14   main()
```

Argument specjalny `args` występuje w nagłówku funkcji po argumentach pozycyjnych i opcjonalnych argumentach pozycyjnych. Nazwa `args` jest zwyczajowo stosowaną nazwą będącą skrótem od „*arguments*” i jest to zazwyczaj zmienna, która zawiera krotkę argumentów pozycyjnych

Na listingu 5.15 przedstawiamy ogólną postać definicji funkcji, w której dwa pierwsze argumenty są pozycyjne, z których drugi jest opcjonalny dzięki przypisanej mu wartości domyślnej, a trzeci jest argumentem specjalnym. Wywołanie funkcji z jednym argumentem (linia 2) spowoduje, że parametry aktualne wywołania dopasowane zostaną od lewej do argumentów i w ten sposób argument `a` przyjmie wartość 1, argument `b` będzie miał ustawioną swoją wartość domyślną, a argument specjalny `args` będzie pusty. W przypadku wywołania funkcji z dwoma argumentami (linia 3), dwa pierwsze argumenty funkcji przyjmą odpowiednio w kolejności występowania od lewej wartości parametrów aktualnych, a argument specjalny będzie nadal pustą krotką. Każde wywołanie z większą niż dwa liczbą parametrów formalnych, spowoduje począwszy od trzeciego parametru wywołania, gromadzenie ich w krotce argumentu specjalnego.

Listing 5.15: *Ogólna postać definicji funkcji z argumentami nienazwanymi*

```
def main():
  func(1)
  func(1, 2)
  func(1, 2, 3, 4, 5)
def func(a, b = 9, *args):
  # a jest obowiązkowym argumentem pozycyjnymi
  print("a =", a)
  # b jest opcjonalnym argumentem pozycyjnym
  print("b =", b )
  # args jest krotką nienazwanych
  # argumentów pozycyjnych
  print("args =", args)
if __name__=='__main__':
  main()
```

W wyniku wykonania programu z listingu 5.15 na ekranie zobaczymy:

```
a = 1
b = 9
args = ()
a = 1
b = 2
args = ()
a = 1
b = 2
args = (3, 4, 5)
```

Próba wywołania funkcji `func` bez argumentów spowoduje wystąpienie błędu:

```
TypeError: func() missing 1 required positional argument: 'a'.
```

5.3. Zbiory

Python posiada wbudowany typ danych dla zbiorów. Zbiór jest nieuporządkowaną i nieindeksowaną kolekcją nie zawierającą duplikatów.

5.3.1. Zbiory - podstawy

Przy pomocy operatora `in` można testować przynależność elementu do zbioru. Zbiory umożliwiają wykonywanie takich podstawowych operacji jak suma, iloczyn, różnica oraz różnica symetryczna. Aby utworzyć zbiór należy użyć nawiasów {} dla sekwencji obiektów

rozdzielonych przecinkami lub funkcji `set` z argumentem będącym obiektem iterowalnym. Przy czym zbiór pusty tworzymy tylko używając funkcji `set` bez argumentów a nie nawiasów {}, ponieważ konstrukcja {} tworzy pusty słownik.

Na listingu 5.16 przedstawiamy różne sposoby tworzenia zbiorów: poprzez ujęcie w nawiasy klamrowe nazw owoców (linie 2-3), wywołując funkcję `set` dla argumentów będących napisami (linie 8 i 9), czy listami liczb (linie 17 i 18) oraz wykonane na tych zbiorach podstawowe działania: sumy (`|`), różnicy (`-`), iloczynu (`&`) oraz różnicy symetrycznej (`^`). Wymienione działania są zgodne z działaniami na zbiorach znanymi z teorii mnogości.

LISTING 5.16: *Zbiory i działania na nich*

```
1  def main():
2    koszyk = {"jabłko", "kiwi", "jabłko",
3              "cytryna", "kiwi", "banan", "mango"}
4    print(koszyk)
5    print("kiwi" in koszyk)        # True
6    print("mandarynka" in koszyk)  # False
7
8    a = set("abracadabra")
9    b = set("alacazam")
10   print("a =", a)
11   print("b =", b)
12   print("a - b =", a - b)   # różnica zbiorów
13   print("a | b =", a | b)   # suma zbiorów
14   print("a & b =", a & b)   # iloczyn zbiorów
15   print("a ^ b =", a ^ b)   # różnica symetryczna zbiorów
16
17   a = set([1,2,3,4,5,6,7,8,9,0])
18   b = set([1,1,2,2,3,3,4,4,5,5,9,9])
19   print("a =", a)
20   print("b =", b)
21   print("a - b =", a - b)   # różnica zbiorów
22   print("a | b =", a | b)   # suma zbiorów
23   print("a & b =", a & b)   # iloczyn zbiorów
24   print("a ^ b =", a ^ b)   # różnica symetryczna zbiorów
25
26 if __name__ == "__main__":
27   main()
```

W wyniku wykonania programu z listingu 5.16 na ekranie zobaczymy:

```
{'cytryna', 'banan', 'jabłko', 'mango', 'kiwi'}
True
```

```
False
a = {'a', 'c', 'r', 'b', 'd'}
b = {'m', 'a', 'c', 'z', 'l'}
a - b = {'d', 'b', 'r'}
a | b = {'m', 'a', 'c', 'r', 'z', 'l', 'b', 'd'}
a & b = {'a', 'c'}
a ^ b = {'z', 'l', 'b', 'm', 'd', 'r'}
a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
b = {1, 2, 3, 4, 5, 9}
a - b = {0, 8, 6, 7}
a | b = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
a & b = {1, 2, 3, 4, 5, 9}
a ^ b = {0, 6, 7, 8}
```

Z kolei na listingu 5.17 przedstawiamy ponownie koszyk owoców reprezentowany zbiorem ich nazw (linie 2-3), który wypisywany jest na ekran, a następnie zostaje przekonwertowany do listy (linia 5). Elementy listy wypisywane są na ekran (linia 6), aby zobaczyć (linie 6 i 8) ich kolejność przed i po sortowaniu (linia 7). Ze zbioru o nazwie `koszyk` usuwane są wszystkie jego elementy przy pomocy metody `clear`, aby z kolei w pętli `for` każdy element listy wstawić do zbioru `koszyk` używając metody `add` (linie 10-11). Po wypisaniu elementów zbioru na ekran, okazuje się, że ich kolejność jest inna niż kolejność wstawianych do zbioru elementów listy ułożonych zgodnie z porządkiem rosnącym. W Pythonie bowiem zbiory są niezmiennymi i nieuporządkowanymi kolekcjami.

Listing 5.17: *Zbiory i działania na nich*

```
1  def main():
2    koszyk = {"jabłko", "kiwi", "jabłko",\
3              "cytryna", "kiwi", "banan", "mango"}
4    print("1:", koszyk)
5    lista = list(koszyk)
6    print("2:", lista)
7    lista.sort()  # sortowanie listy
8    print("3:", lista)
9    koszyk.clear()
10   for owoc in lista:
11     koszyk.add(owoc)
12   print("4:", koszyk)
13   koszyk.clear()
14
15 if __name__ == "__main__":
16   main()
```

W wyniku wykonania programu z listingu 5.17 na ekranie pojawią się informacje:

```
1: {'banan', 'cytryna', 'mango', 'kiwi', 'jabłko'}
2: ['banan', 'cytryna', 'mango', 'kiwi', 'jabłko']
3: ['banan', 'cytryna', 'jabłko', 'kiwi', 'mango']
4: {'banan', 'cytryna', 'mango', 'kiwi', 'jabłko'}
```

5.3.2. Metody klasy `set`

W tym podrozdziale wymienimy tylko metody klasy `set` bez podawania przykładów ich użycia. Czytelnika zachęcamy do przetestowania ich w programach realizujących treści zadań do samodzielnego rozwiązania.

- `a.add(x)` – Dodaje obiekt `x` do zbioru `a`.
- `a.clear()` – Usuwa wszystkie elementy ze zbioru `a`.
- `a.copy()` – Zwraca płytką kopię zbioru `a` (kopiuje tylko referencje do obiektów będących elementami zbioru).
- `a.discard(elem)` – Usuwa element ze zbioru `a`, jeżeli należy on do tego zbioru. W przeciwnym przypadku nic nie robi.
- `a.difference(*others)` – `a - other_1 - other_2 - ...`
  Zwraca różnicę dwóch lub więcej zbiorów jako nowy zbiór. (W nowym zbiorze będą wszystkie elementy zbioru `a`, które nie należą do pozostałych zbiorów.)
- `a.difference_update(*others)` – `a -= other_1 | other_2 | ...`
  Usuwa ze zbioru `a` wszystkie elementy należące do pozostałych zbiorów.
- `a.intersection(*others)` – `a & other_1 & other_2 & ...`
  Zwraca część wspólną zbioru `a` oraz pozostałych zbiorów jako nowy zbiór.
- `a.intersection_update(*others)` – `a &= other_1 & other_2 & ...`
  Modyfikuje zbiór `a` pozostawiając tylko te jego elementy, które należą również do pozostałych zbiorów.
- `a.isdisjoint(other)` – `a != other`
  Zwraca wartość `True`, jeśli oba zbiory są rozłączne.
- `a.issubset(other)` – `a <= other`
  Zwraca wartość `True`, jeżeli zbiór `a` zawiera się w zbiorze `other`.
- `a.issuperset(other)` – `a >= other`
  Zwraca wartość `True`, jeżeli zbiór `a` zawiera zbiór `other`.
- `a.pop()` – Usuwa jeden element ze zbioru `a` i zwraca jego wartość. Wyrzuca wyjątek `KeyError`, jeżeli zbiór jest pusty.
- `a.remove(elem)` – Usuwa podany element ze zbioru `a`. Jeżeli element ten nie należy do zbioru, to wyrzuca wyjątek `KeyError`.

- `a.symmetric_difference(other)` – `a ^ other`
  Zwraca różnicę symetryczną dwóch zbiorów jako nowy zbiór.
- `a.symmetric_difference_update(other)` – `a ^= other`
  Modyfikuje zbiór `a` jako różnicę symetryczną tego zbioru i zbioru `other`.
- `a.union(*others)` – `a | other_1 | other_2 | ...`
  Zwraca sumę zbioru `a` oraz pozostałych zbiorów jako nowy zbiór.
- `a.update(*others)` – `a |= other_1 | other_2 | ...`
  Modyfikuje zbiór `a` jako sumę tego zbioru i pozostałych zbiorów.

5.3.3. Zbiory zamrożone

W Pythonie występuje specjalny typ zbioru nazywany **zbiorem zamrożonym**, który jest obiektem klasy `frozenset`. Zbiory zamrożone to zbiory, których nie można modyfikować: ani nie można dodawać do nich nowych elementów, ani z nich elementów usuwać. Mogą być one elementami innych zbiorów, jak i zbiorów zamrożonych, ponieważ są jednocześnie niemutowalne oraz haszowalne. Zbiory zamrożone można utworzyć przy użyciu funkcji `frozenset`, której argumentem może być dowolny obiekt iterowalny. Wywołanie funkcji `frozenset` bez argumentów tworzy pusty zbiór zamrożony. Dla zbiorów zamrożonych dostępne są metody: `copy`, `difference`, `intersection`, `isdisjoint`, `issubset`, `issuperset`, `symmetric_difference` oraz `union`.

Przedstawimy teraz kilka przykładów działań na dwóch zbiorach zamrożonych utworzonych z elementów list (sekwencji):

```
>>> A = frozenset([1,2,3,4])
>>> B = frozenset([1,2,5,6])
```

- Sprawdzenie, czy zbiory `A` i `B` są rozłączne: `isdisjoint`:

  ```
  >>> A.isdisjoint(B)
  False
  ```

- Obliczenie różnicy zbiorów `A` i `B`:

  ```
  >>> A.difference(B)
  frozenset({3, 4})
  ```

- Obliczenie sumy zbiorów `A` i `B`:

  ```
  >>> A | B
  frozenset({1, 2, 3, 4, 5, 6})
  ```

- Obliczenie różnicy symetrycznej zbiorów `A` i `B`:

  ```
  >>> A ^ B
  frozenset({3, 4, 5, 6})
  ```

- Obliczenie części wspólnej zbiorów zbiorów `A` i `B`:

```
>>> A & B
frozenset({1, 2})
```

Próba dodania elementu do zbioru zamrożonego, który jest niemutowalny, powoduje wystąpienie błędu `AttributeError`:

```
>>> A.add(11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Ponieważ zbiory zamrożone są haszowalne, można względem nich stosować funkcję `hash`:

```
>>> hash(A)
5575258175646371796
>>> hash(B)
2799569190866510694
```

5.4. Słowniki

Jednym z wbudowanych typów w Pythonie są `słowniki` (ang. *dictionary*). Określają one wzajemną relację między kluczem a wartością. Słownik jest zmiennym (modyfikowalnym), nieposortowanym zestawem par `klucz:wartość` ujętym w nawiasy klamrowe (`{}`). Słowniki można porównać do „pamięci asocjacyjnej" lub „tablic asocjacyjnych" z innych języków programowania, czy klucza głównego w relacyjnych bazach danych, który pozwala zidentyfikować pojedynczy wiersz w tabeli. Słowniki wewnętrznie są implementowane jako tablice haszujące dodatkowo optymalizowane, co powoduje bardzo szybkie wydobywanie z nich danych.

Słowniki indeksowane są kluczami, które muszą być obiektami niemutowalnymi, np. takimi jak liczby czy napisy. Krotki mogą zostać użyte również jako klucze, jeżeli ich składowymi są liczby, napisy, krotki i zbiory zamrożone (ang. *frozenset*). Nie można użyć zwykłych list jako kluczy, ponieważ można je modyfikować choćby za pomocą metody `append`.

5.4.1. Tworzenie słownika

Utworzenie pustego słownika wymaga albo przypisania do zmiennej `d` pustych nawiasów klamrowych (`{}`):

```
>>> d = {}
>>> d
{}
```

albo użycia funkcji wbudowanej `dict`:

```
>>> d = dict{}
>>> d
{}
```

Można utworzyć słownik z elementami będącymi parami `klucz:wartość` rozdzielonymi przecinkami i ograniczonymi nawiasami klamrowymi, np.

```
>>> d = {"a":"ala", "b":"beata"}
>>> d
{'a': 'ala', 'b': 'beata'}
```

W tym przypadku zarówno klucze, jak i wartości są łańcuchami.

Teraz zastosujemy funkcję `dict`, aby uzyskać ten sam słownik.

```
>>> d = dict("a"="ala", "b"="beata")
>>> d
{'a': 'ala', 'b': 'beata'}
```

Na listingu 5.18 przedstawiamy różne sposoby tworzenia słowników:

- w standardowy sposób - linie 2 i 3;
- z użyciem funkcji `zip` (linia 4) - funkcja ta łączy ze sobą elementy z różnych obiektów iterowalnych, takich jak listy, krotki, zbiory i zwraca `iterator` (więcej informacji o iteratorach odnajdzie czytelnik w podrozdziale 8.4). Możemy jej użyć do połączenia ze sobą dwóch list, aby utworzyć pary wstawiane do słownika;
- z użyciem listy dwuelementowych krotek - linia 5;
- z użyciem innego słownika - linia 6.

LISTING 5.18: *Tworzenie słowników*

```
1  def main():
2    a = dict(one=1, two=2, three=3)
3    b = {'one': 1, 'two': 2, 'three': 3}
4    c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
5    d = dict([('two', 2), ('one', 1), ('three', 3)])
6    e = dict({'three': 3, 'one': 1, 'two': 2})
7    print(a, b, c, d, e, sep='\n')
8    print(a == b == c == d == e)
9  if __name__=='__main__':
10   main()
```

Element w słowniku identyfikowany jest poprzez `klucz`. Zatem, aby uzyskać wartość elementu, należy użyć dla zmiennej słownikowej nawiasów kwadratowych i w środku podać klucz identyfikujący element:

```
>>> d["a"]
'ala'
>>> d["b"]
'beata'
```

Można dostać się do wartości za pomocą klucza, ale nie można dostać się do klucza za pomocą wartości. Tak więc `d['b']` zwraca `'beata'`, ale wywołanie `d['beata']` sprawi, że wystąpi wyjątek, ponieważ `'beata'` nie jest kluczem słownika `d`.

```
>>> d["ala"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'ala'
```

5.4.2. Dodawanie i nadpisywanie danych w słowniku

Ponieważ klucze w słowniku nie mogą się powtarzać, przypisując wartość do istniejącego klucza, będziemy nadpisywać starszą wartość.

```
>>> d['b'] = 'barbara'
>>> d
{'a': 'ala', 'b': 'barbara'}
```

W każdej chwili do słownika możemy dodać nową parę `klucz:wartość`. Składnia dodawania pary do słownika jest identyczna z tą, która modyfikuje istniejącą wartość. Niestety czasami może to spowodować problem, gdyż możemy myśleć, że dodaliśmy nową wartość do słownika, gdy w rzeczywistości nadpisaliśmy już istniejącą.

```
>>> d['c'] = 'czeslaw'
>>> d
{'a': 'ala', 'b': 'beata', 'c': 'czeslaw'}
```

Zauważmy, że słownik nie przechowuje kluczy w sposób posortowany. W Pythonie od wersji 3.6 kolejność par `klucz:wartość` jest zgodna z kolejnością ich wstawienia do słownika (w poprzednich wersjach Pythona kolejność pozycji była nieprzewidywalna). Należy pamiętać, że nazwy kluczy są wrażliwe na wielkość liter, co pokazujemy w poniższym przykładzie.

```
>>> d = {}
>>> d["klucz"] = "wartosc"
>>> d["klucz"] = "inna wartosc"
>>> d
{'klucz': 'inna wartosc'}
>>> d["Klucz"] = "jeszcze inna wartosc"
>>> d
{'klucz': 'inna wartosc', 'Klucz': 'jeszcze inna wartosc}
```

Słowniki nie są przeznaczone tylko dla łańcuchów znaków. Wartość w słowniku może być dowolnym typem danych: łańcuchem znaków, liczbą całkowitą, obiektem, a nawet innym słownikiem. W pojedynczym słowniku wszystkie wartości nie muszą być tego samego typu; możemy wstawić do niego obiekty mieszanych typów danych, np. łańcuchów znaków i liczb całkowitych.

```
>>> d
{'a': 'ala', 'b': 'beata', 'c': 'czesław'}
>>> d['c'] = 3
>>> d
{'a': 'ala', 'b': 'beata', 'c': 3}
```

Klucze w słowniku są bardziej restrykcyjne, ale mogą być łańcuchami znaków, liczbami całkowitymi i kilkoma innymi typami, które są niezmienne. Przy czym klucze wewnątrz jednego słownika nie muszą posiadać tego samego typu, np. mogą być łańcuchami znaków, jak i liczbami całkowitymi.

```
>>> d[42] = "weronika"
>>> d
{'a': 'ala', 'b': 'beata', 'c': 3, 42: 'weronika'}
```

5.4.3. Usuwanie elementów ze słownika

Aby usunąć określony element ze słownika, który jest wskazywany przez podany klucz, używamy instrukcji `del`.

```
>>> d
{'a': 'ala', 'b': 'beata', 'c': 3, 42: 'weronika'}
>>> del d[42]
>>> d
{'a': 'ala', 'b': 'beata', 'c': 3}
```

Próba usunięcia pozycji wskazywanej przez klucz, którego nie ma w słowniku, skutkuje wystąpieniem błędu `KeyError`.

```
>>> del(d[42])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 42
```

5.4.4. Metody wbudowane słowników

Do podstawowych operacji wykonywanych na słownikach należą:

- Sprawdzanie, czy klucz jest w słowniku przez użycie operatora `in`.

  ```
  >>> Dict = {1:'a',2:'b',3:'c',4:'d',5:'e'}
  >>> 5 in Dict
  True
  ```

- `Dict.clear()` nie zwraca wartości, usuwa wszystkie pary `klucz:wartość` ze słownika.

  ```
  >>> Dict.clear()
  >>> Dict
  {}
  ```

- `Dict.copy()` zwraca kopię słownika `Dict`.

  ```
  >>> Dict = {1:'a',2:'b',3:'c',4:'d',5:'e'}
  >>> Dict_copy = Dict.copy()
  >>> Dict_copy
  {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
  >>> Dict[2]='zzz'
  >>> Dict
  {1: 'a', 2: 'zzz', 3: 'c', 4: 'd', 5: 'e'}
  >>> Dict_copy
  {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
  ```

- `Dict.fromkeys(S[,v])` - parametr `v` jest opcjonalny - tworzy słownik `Dict` z kluczami z `S` i wartościami równymi `v`.

  ```
  >>> Dict_fromkeys = {}
  >>> S = 'abcd'
  >>> Dict_fromkeys = Dict_fromkeys.fromkeys(S,10)
  >>> Dict_fromkeys
  {'a': 10, 'b': 10, 'c': 10, 'd': 10}
  ```

- `Dict.get(k[,d])` - parametr `d` jest opcjonalny - zwraca `Dict[k]` jeśli `k` jest kluczem w słowniku, w przeciwnym razie zwraca `d`. Jeżeli wywołano tę metodę z jednym

tylko argumentem - kluczem i takiego klucza nie ma w słowniku, zostanie zwrócona wartość `None`.

```
>>> Dict_fromkeys
{'a': 10, 'b': 10, 'c': 10, 'd': 10}
>>> Dict_fromkeys.get('a')
10
>>> Dict_fromkeys.get('aa')
>>> Dict_fromkeys.get('aa',0)
0
```

- `Dict.pop(klucz[,błąd])` - parametr **błąd** jest opcjonalny - zwraca wartość elementu korespondującego z kluczem i usuwa go ze słownika, jeśli klucza nie ma w słowniku zwraca wartość **błąd**, a jeśli wartość **błąd** nie została podana „zwraca" `KeyError`.

```
>>> Dict
{1: 'a', 2: 'zzz', 3: 'c', 4: 'd', 5: 'e'}
>>> Dict.pop(1)
'a'
>>> Dict.pop(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> Dict.pop(1,"Nie ma w słowniku takiego klucza")
'Nie ma w słowniku takiego klucza'
```

- `Dict.popitem()` zwraca parę (**klucz**, **wartość**) i usuwa ją ze słownika, a jeśli słownik jest pusty „zwraca" błąd `KeyError`.

```
>>> Dict.popitem()
(5, 'e')
>>> Dict.popitem()
(4, 'd')
>>> Dict.popitem()
(3, 'c')
>>> Dict.popitem()
(2, 'zzz')
>>> Dict.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

- `Dict.setdefault(klucz[,błąd])` - parametr `błąd` jest opcjonalny - zwraca wartość elementu korespondującego z kluczem, jeśli klucza nie ma w słowniku zwraca wartość `błąd` i wstawia do słownika parę (`klucz:wartość`).

```
>>> Dict = {1:'a',2:'b',3:'c',4:'d',5:'e'}
>>> Dict.setdefault(1)
'a'
>>> Dict.setdefault(11)
>>> Dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 11: None}
>>> Dict.setdefault(12,"Nie ma w słowniku")
'Nie ma w słowniku'
>>> Dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e',
 11: None, 12: 'Nie ma w słowniku'}
```

- `Dict.update([other])` nie zwraca wartości, aktualizuje słownik `Dict` parami `klucz:wartość` ze słownika podanego jako argument lub innego obiektu iterowalnego składającego się z par (`klucz,wartość`).

```
>>> Dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 11: None,
 12: 'Nie ma w słowniku'}
>>> Dict1
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
>>> Dict.update(Dict1)
>>> Dict
{1: 'a', 2: 'xxx', 3: 'c', 4: 'd', 5: 'e', 11: None,
12: 'Nie ma w słowniku', 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
>>> Dict = {'x':2}
>>> Dict.update([('y',3),('z',4)])
>>> Dict
{'x': 2, 'y': 3, 'z': 4}
>>> Dict.update(imie = 'Ala',nazwisko = 'XXX')
>>> Dict
{'x': 2, 'y': 3, 'z': 4, 'imie': 'Ala', 'nazwisko': 'XXX'}
```

- `Dict.values()` zwraca listę wartości słownika.

```
>>> Dict.values()
dict_values([2, 3, 4, 'Ala', 'XXX'])
```

5.4.5. Argumenty nazwane funkcji

Na końcu listy argumentów funkcji można opcjonalnie użyć formy specjalnej `**kwargs`. Jest to słownik (ewentualnie pusty), do którego wstawiane będą wszelkie dodatkowe nazwane argumenty funkcji w postaci par `klucz:wartość`, gdzie kluczem jest nazwa argumentu, a wartością - wartość tego argumentu.

LISTING 5.19: *Argumenty podawane za pomocą słów kluczowych*

```
def main():
  d = dict(jan = 31)
  print('1:',d)
  d = dict(jan = 31, feb = 28)
  print('2:',d)
  d = dict(jan = 31, feb = 28, mar = 31)
  print('3:',d)
  d = dict(jan = 31, feb = 28, mar = 31, apr = 30)
  print('4:',d)
if __name__=='__main__':
  main()
```

W wyniku wykonania programu z listingu 5.19 powstaje za każdym razem słownik z argumentów podawanych za pomocą słów kluczowych, a na ekranie pojawiają się informacje o ich zawartości:

```
1: {'jan': 31}
2: {'jan': 31, 'feb': 28}
3: {'jan': 31, 'feb': 28, 'mar': 31}
4: {'jan': 31, 'feb': 28, 'mar': 31, 'apr': 30}
```

Na listingu 5.20 pokazujemy zastosowanie formy specjalnej argumentów `**kwargs` w funkcji `pokaz_sume`. Dzięki temu możemy wywoływać tę funkcję z dowolną liczbą argumentów nazwanych.

LISTING 5.20: *Dowolna liczba argumentów nazwanych*

```
def main():
  pokaz_sume(1, 2, a = 5, b = 8)
def pokaz_sume(x, y, **inne):
  print( "x: {0}, y: {1}".format(x,y), end='')
  print("\nWartości w 'inne': {}".format(list(inne.values())))
  suma = x + y + sum(inne.values())
  print("Suma wartości argumentów wynosi: {}".format(suma))
if __name__=='__main__':
  main()
```

W wyniku wykonania programu z listingu 5.20 widzimy na ekranie wartości argumentów pozycyjnych, słownika `inne` oraz obliczoną sumę ich wszystkich.

```
x: 1, y: 2
Wartości w 'inne': [5, 8]
Suma wartości argumentów wynosi: 16
```

Ogólna i pełna postać listy argumentów w definicji funkcji została przedstawiona na listingu 5.21.

Listing 5.21: *Ogólna postać definicji funkcji*

```
def main():
  func(1, 2, 3, 4, 5, c = 5, k1 = 11, k2 = 12)
  func(1, 2, 3, 4, 5, c = 5, d = 7, k1 = 11, k2 = 12)
  func(1, 2, 3, 4, 5, c = 5, k1 = 11, k2 = 12, d = 7)

def func(a, b, *args, c, d = 6, **kwargs):
  # a i b są obowiązkowymi argumentami pozycyjnymi
  print("a =", a, "b =", b )
  # args jest krotką nienazwanych argumentów pozycyjnych
  print("args =", args)
  # c i d są argumentami, które są przekazywane
  # wyłącznie  za pomocą słów kluczowych
  print("c =", c, "d =", d)
  # kwargs jest słownikiem
  print("kwargs =", kwargs, "\n")

if __name__=='__main__': main()
```

W wyniku wykonania programu z listingu 5.21 na ekranie pokażą się informacje:

```
a = 1 b = 2
args = (3, 4, 5)
c = 5 d = 6
kwargs = {'k1': 11, 'k2': 12}
a = 1 b = 2
args = (3, 4, 5)
c = 5 d = 7
kwargs = {'k1': 11, 'k2': 12}
a = 1 b = 2
args = (3, 4, 5)
c = 5 d = 7
kwargs = {'k1': 11, 'k2': 12}
```

5.5. Zadania do samodzielnego rozwiązania

5.5.1. Listy

1. Zdefiniuj funkcję `F1(L)`, która zwróci ilość liczb parzystych znajdujących się w liście `L` przekazanej jako argument funkcji. Przetestuj w funkcji `main` działanie funkcji `F1` dla listy 100 liczb całkowitych wygenerowanych losowo z przedziału `⟨1,100⟩`.
2. Zdefiniuj funkcję `F2(L)`, która wypisze na ekranie ile razy w liście `L` występuje każda z cyfr (w tym celu wykorzystaj pomocniczą listę, której indeksy reprezentują cyfry, a wartości liczbę ich wystąpień). Przetestuj w funkcji `main` działanie funkcji `F2` dla listy 100 liczb całkowitych wygenerowanych losowo z przedziału `⟨-50,50⟩`.
3. Zdefiniuj funkcję `F3(L)`, która obliczy i zwróci średnią arytmetyczną elementów listy `L`. Przetestuj w funkcji `main` działanie funkcji `F3` dla listy 100 liczb całkowitych wygenerowanych losowo z przedziału `⟨10,20⟩`.
4. Zdefiniuj funkcję `F4(L)`, która odnajdzie w liście `L` element najmniejszy i zwróci ostatni indeks, pod którym znajduje się ten element i jego wartość. Przetestuj w funkcji `main` działanie funkcji `F4` dla listy 100 liczb całkowitych wygenerowanych losowo z przedziału `⟨-50,50⟩`.
5. Zdefiniuj funkcję `F5(L)`, która zwróci liczbę samogłosek znajdujących się w liście `L`. Przetestuj w funkcji `main` działanie funkcji `F5` dla listy 100 znaków wygenerowanych losowo spośród małych liter alfabetu.
6. Zdefiniuj funkcję `F6(L)`, która obliczy i zwróci różnicę między największym i najmniejszym elementem listy `L` przekazanej jako argument funkcji. Przetestuj w funkcji `main` działanie funkcji `F6` dla listy 1000 liczb całkowitych wygenerowanych losowo z przedziału `<1,100>`.
7. Zdefiniuj funkcję `F7(L, n)`, która zwróci `True` jeśli na liście `L` znajduje się n-ta potęga liczby 2; w przeciwnym przypadku funkcja zwróci `False`. Przetestuj w funkcji `main` działanie funkcji `F7` dla listy 100 liczb całkowitych wygenerowanych losowo z przedziału `<1,64>` oraz n wybranego losowo z przedziału `⟨0,6⟩`.
8. Zdefiniuj funkcję `F8(L)`, która utworzy i zwróci listę składającą się z niepowtarzających się elementów listy `L`. Przetestuj w funkcji `main` działanie funkcji `F8` dla listy 100 liczb całkowitych wygenerowanych losowo z przedziału `⟨-50,50⟩`.
9. Zdefiniuj funkcję `F9(L, K)`, która utworzy i zwróci listę składającą się ze wspólnych elementów list `L` i `K`. Przetestuj w funkcji `main` działanie funkcji `F9` dla dwóch list 100 liczb całkowitych wygenerowanych losowo z przedziału `⟨-50,50⟩`.
10. Zdefiniuj funkcję `F10(L, K)`, która zwróci `True` jeżeli listy `L` i `K` mają tyle samo i takie same elementy, `False` w przeciwnym przypadku. Przetestuj w funkcji `main` działanie funkcji `F10` dla dwóch list w taki sposób, aby uwzględnić wszystkie przy-

padki (gdy listy nie są tej samej długości, gdy mają tę samą długość i różne elementy oraz gdy mają tę samą długość i te same elementy w różnej kolejności).

11. Dla każdego z podpunktów przyjmujemy, że mamy początkowo pustą listę `L=[]`, którą wypełniamy odpowiednimi liczbami z użyciem jednej lub dwóch instrukcji iteracyjnych `for` i wyświetlamy na ekranie:
    (a) `1 2 3 4 5 6 7 8 9 10`
    (b) `0 2 4 6 8 10 12 14 16 18 20`
    (c) `1 4 9 16 25 36 49 64 81 100`
    (d) `0 0 0 0 0 0 0 0 0 0`
    (e) `0 1 0 1 0 1 0 1 0 1`
    (f) `0 1 2 3 4 0 1 2 3 4`
    (g) `1 -2 3 -4 5 -6 7 -8 9 -10`
    (h) `0 1 0 4 0 9 0 16 0 25 0 36 0 49 0 64 0 81 0 100`
12. Zdefiniuj funkcję `ile_ujemnych`, która zwraca jako swój wynik liczbę ujemnych elementów listy podanej jako jedyny argument tej funkcji. Przetestuj tę funkcję w funkcji `main`.
13. Zdefiniuj funkcję `iloczyn`, która zwraca jako swój wynik iloczyn liczb będących elementami listy podanej jako jedyny argument tej funkcji. Przetestuj tę funkcję w funkcji `main`.
14. Zdefiniuj funkcję `minmax`, która zwraca jako swój wynik krotkę dwóch liczb, z których pierwsza to minimum a druga to maksimum z listy podanej jako jedyny argument tej funkcji. Przykładowo, dla listy:

    ```
    a = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    ```

    funkcja `minmax(a)` powinna zwrócić krotkę `(28,31)`. Przetestuj tę funkcję w funkcji `main`.
15. Zdefiniuj funkcję obliczającą i zwracającą sumę naprzemienną wszystkich elementów listy. Przykładowo, dla listy `[1, 4, 16, 9, 9, 7, 4, 9, 11]` funkcja powinna obliczyć i zwrócić jako wynik:

    ```
    1 - 4 + 16 - 9 + 9 - 7 + 4 - 9 + 11 = 12
    ```

    Przetestuj tę funkcję w funkcji `main`.
16. Napisz program, który wczytuje kolejną liczbę i dodaje ją do listy, o ile nie znajduje się już w liście. Gdy lista zawiera dziesięć liczb, program wyświetla jej zawartość i kończy pracę.
17. Napisz program implementujący sito Eratostenesa, który dodaje wszystkie liczby od 2 do 1000 do początkowo pustej listy, po czym usuwa wielokrotności 2 (ale nie 2), wielokrotności 3 (ale nie 3) i tak dalej, aż do wielokrotności 100. Następnie program wypisuje liczby pozostałe w liście.

18. Dla każdego z poniższych punktów napisz funkcję, która wykonuje odpowiednie zadanie dla listy liczb całkowitych będącej jej argumentem. Przetestuj każdą funkcję w funkcji `main`.
    (a) Zamienia miejscami pierwszy i ostatni element na liście.
    (b) Przesuwa wszystkie elementy o jedną pozycję w prawo, a ostatni element przenosi na początek listy, np. lista `[1 4 9 16 25]` ma być przekształcona w listę `[25 1 4 9 16]`.
    (c) Zastępuje wszystkie parzyste elementy listy liczbą 0.
    (d) Zastępuje każdy element listy, z wyjątkiem pierwszego i ostatniego, przez większy z dwóch sąsiednich elementów.
    (e) Usuwa element środkowy, jeśli długość listy jest nieparzysta lub dwa środkowe elementy, jeśli długość jest parzysta.
    (f) Przenosi wszystkie parzyste elementy listy na jej początek, z zachowaniem kolejności ich występowania.
    (g) Zwraca drugi co do wielkości element na liście.
    (h) Zwraca `True`, jeśli lista jest posortowana w porządku rosnącym. W przeciwnym przypadku zwraca `False`.
    (i) Zwraca `True`, jeśli lista zawiera jakiekolwiek dwa sąsiadujące równe elementy. W przeciwnym przypadku zwraca `False`.
    (j) Zwraca `True`, jeśli lista zawiera jakiekolwiek dwa równe elementy (które nie muszą znajdować się obok siebie). W przeciwnym przypadku zwraca `False`.
19. Dana jest lista zawierająca 100 wygenerowanych losowo liczb całkowitych z przedziału `<-50,50>`. Napisz program, który wyznaczy średnią arytmetyczną, medianę, dominantę i odchylenie standardowe elementów znajdujących się w tej liście. Jaki procent elementów listy znajduje się w przedziale $< \overline{x} - \sigma, \overline{x} + \sigma >$, gdzie $\overline{x}$ oznacza średnią arytmetyczną, a σ - odchylenie standardowe?
20. Dane są listy `L1` i `L2` zawierające cyfry pewnych liczb z systemu trójkowego. Napisz program umożliwiający dodawanie tych liczb (bez użycia konwersji na system dziesiętny).
21. Dana jest lista zawierająca zarobki pracowników pewnej firmy. Napisz program, który zaktualizuje dane na liście w taki sposób, aby każdy z pracowników otrzymał `10%` podwyżki.
22. Dane są dwie listy: `Lo`, `Lw`. Napisz program, który na podstawie ocen znajdujących się na liście `Lo` oraz odpowiadających im wag z listy `Lw`, wyznaczy ocenę, która jest średnią ważoną zestawionych danych.
23. Utwórz listę zawierającą wszystkie liczby, które można utworzyć z cyfr `1`, `2`, `3`, `4`.
24. Napisz program, który utworzy listę składającą się z list będących permutacjami `5`-elementowego zbioru. Zadbaj o losowy zestaw danych tego zbioru.

25. Zdefiniuj funkcję `PN`, która zwraca jako swój wynik krotkę dwóch liczb, z których pierwsza stanowi ilość parzystych, a druga - ilość nieparzystych elementów znajdujących się w liście przekazanej jako jedyny argument tej funkcji.
26. Napisz funkcję `ile(L,a,b)`, która zwróci informację, ile liczb naturalnych z przedziału `(a,b)` znajduje się w liście `L`.
27. Dana jest lista zawierająca wyrazy pewnego skończonego ciągu. Napisz program, który sprawdza, czy jest to ciąg:
    (a) monotoniczny,
    (b) arytmetyczny.

5.5.2. Zbiory

1. Załóżmy, że dane są trzy zbiory: `set1`, `set2` i `set3` liczb wygenerowanych losowo z przedziału `<1,15>` o różnej długości. Napisz instrukcje w jednym programie, które:
    (a) Utworzą zbiór tych wszystkich elementów, które należą do `set1` lub `set2`, ale nie należą do obydwu jednocześnie.
    (b) Utworzą zbiór tych wszystkich elementów, które należą tylko do jednego z trzech zbiorów `set1`, `set2` i `set3`.
    (c) Utworzą zbiór wszystkich elementów, które należą do dokładnie dwóch zbiorów spośród `set1`, `set2` i `set3`.
    (d) Utworzą zbiór wszystkich tych liczb całkowitych w zakresie od 1 do 25, które nie należą do `set1`.
    (e) Utworzą zbiór wszystkich tych liczb całkowitych w zakresie od 1 do 25, które nie należą do któregoś z trzech zbiorów `set1`, `set2` i `set3`.
    (f) Utworzą zbiór wszystkich tych liczb całkowitych w zakresie od 1 do 25, które nie należą do żadnego z trzech zbiorów `set1`, `set2` i `set3`.
2. Dla każdego z poniższych punktów napisz funkcję, która pobiera jako argumenty dwa łańcuchy znaków i zwraca jako wynik:
    (a) Zbiór składający się z małych oraz wielkich liter, które występują jednocześnie w obu łańcuchach.
    (b) Zbiór składający się z małych oraz wielkich liter, które występują w co najmniej jednym łańcuchu.
    (c) Zbiór składający się z małych oraz wielkich liter, które nie występują w żadnym z łańcuchów.
    (d) Zbiór składający się z wszystkich znaków (innych niż litery) występujących jednocześnie w obu łańcuchach.
3. Zaimplementuj znaną starożytnym Grekom metodę obliczania liczb pierwszych nazywaną sitem Eratostenesa. Metoda ta tworzy zbiór wszystkich liczb pierwszych nie większych od n. Wczytaj liczbę do zmiennej `n`, po czym wstaw do zbioru `primes`

wszystkie liczby od 2 do n. Następnie usuń wszystkie wielokrotności liczby 2 (z wyjątkiem 2), potem wszystkie wielokrotności liczby 3 (z wyjątkiem 3) i tak dalej aż do pierwiastka z liczby n. Po zakończeniu obliczeń wypisz zbiór primes.

4. Zmodyfikuj rozwiązanie poprzedniego zadania poprzez utworzenie funkcji sieve(n), która zwraca jako swój wynik zbiór liczb pierwszych nie większych od n.
5. Utwórz listę zawierającą niepowtarzające się imiona dziesięciu osób, a następnie w sposób losowy utwórz trzy zbiory:
   - zbiór A - zbiór osób, które znają język angielski;
   - zbiór N - zbiór osób, które znają język niemiecki;
   - zbiór F - zbiór osób, które znają język francuski.

   Wyznacz następujące zbiory i wyznacz moc każdego z nich:
   - zbiór osób, które znają co najmniej jeden język (spośród podanych powyżej);
   - zbiór osób, które znają dokładnie dwa języki;
   - zbiór osób, które znają trzy języki;
   - zbiór osób, które znają język angielski, ale nie znają języka niemieckiego.
6. Dany jest zbiór set1 zawierający litery i cyfry. Utwórz zbiory litery oraz cyfry zawierające odpowiednio litery i cyfry występujące w zbiorze set1.
7. Dane są trzy liczby całkowite. Utwórz następujące zbiory:
   - zbiór o elementach będących cyframi występującymi w każdej spośród podanych trzech liczb;
   - zbiór cyfr, które nie występują w żadnej z podanych liczb.
8. Dany jest napis n. Utwórz zbiór zawierający kody ASCII znaków z podanego napisu.
9. Dane są dwie listy zawierające sto wygenerowanych losowo liczb całkowitych z przedziału <0,100>. Utwórz następujące zbiory:
   - zbiór A zawierający elementy występujące w co najmniej jednej z list;
   - zbiór B zawierający wyłącznie elementy wspólne dla obu list;
   - zbiór C zawierający elementy, które występują w obu listach o ile są kwadratami pewnych liczb naturalnych.

5.5.3. Słowniki

1. Dla każdego z poniższych punktów napisz funkcję, która pobiera jako argument łańcuch znaków i zwraca jako wynik:
   (a) Słownik zawierający dla każdego znaku występującego w łańcuchu podanym jako argument informację, ile razy znak ten występuje w tym łańcuchu.
   (b) Słownik zawierający informację, ile razy każda litera występuje w łańcuchu podanym jako argument, o ile w nim występuje chociaż raz, np. dla łańcucha znaków 'abrakadabra' funkcja powinna zwrócić następujący słownik: {'b':2,'r':2,'d':1,'a':5,'k':1} (kolejność par w słowniku jest nieistotna).

(c) Słownik taki, jak w poprzednim zadaniu, lecz utworzony bez rozróżniania małych i wielkich liter.

(d) Literę występującą najczęściej w łańcuchu podanym jako argument. Jeżeli jest kilka takich liter, to funkcja może zwrócić dowolną z nich.

W funkcji `main` przetestuj daną funkcję poprzez czytanie łańcucha ze standardowego wejścia i wywołanie tej funkcji.

2. Napisz program, który wczytuje liczby podawane przez użytkownika, dodaje je do słownika i określa liczbę ich wystąpień (liczby stanowią klucze, a liczby ich wystąpień – wartości). W razie podania elementu niebędącego liczbą całkowitą program ma wypisać odpowiedni komunikat i kontynuować działanie. Po naciśnięciu klawisza `<Enter>`, bez uprzedniego podania wartości, program ma wypisać słownik i zakończyć działanie.
3. Napisz program, który czyta kolejne linie ze standardowego wejścia i rozkłada linie na słowa za pomocą metody `split()`. Następnie dla otrzymanej listy elementów sprawdza, czy kolejny element jest liczbą – jeżeli jest, to dodaje go do słownika. Program kończy wczytywanie po wczytaniu pustej linii. Program ma wypisać słownik, w którym kluczami są liczby, a wartościami – liczby ich wystąpień.
4. Zmodyfikuj powyższy program tak, aby dla otrzymanej listy elementów sprawdzał, czy dany element jest trzycyfrową liczbą dodatnią i jeśli tak - dodawał go do słownika.
5. Napisz funkcję, która dla łańcucha znaków przekazanego jako argument zwraca:
   (a) słownik zawierający samogłoski wraz z ilością wystąpień;
   (b) słownik zawierający polskie znaki diakrytyczne wraz z ilością ich wystąpień;
   (c) słownik zawierający znaki inne niż alfanumeryczne wraz z ilością ich wystąpień.

5.5.4. Zadania różne

(a) Napisz funkcję `mediana(*liczby)`, która zwróci medianę liczb podanych jako argumenty.

(b) Napisz funkcję `wypisz(*args)`, która wypisuje w kolejnych liniach swoje argumenty w porządku alfanumerycznym.

(c) Napisz funkcję `kwsum(*liczby)`, która zwróci kwadrat sumy przekazanych argumentów.

(d) Napisz funkcję `sumkw(*liczby)`, która zwróci sumę kwadratów liczb przekazanych jako argumenty.

Rozdział 6

Obsługa wyjątków i plików

W rozdziale tym przedstawimy, w jaki sposób w Pythonie radzić sobie z sytuacjami wyjątkowymi tak, aby program mógł nadal kontynuować swoje działanie. Ponadto opiszemy proces czytania i zapisu danych do pliku przy pomocy odpowiednich funkcji wbudowanych przeznaczonych do realizacji tego rodzaju zadań.

6.1. Błędy składniowe

Najbardziej powszechnym rodzajem błędów, szczególnie na początku nauki, są błędy składniowe, znane również jako błędy „parsingu". Na standardowym wyjściu parser powtarza błędny wiersz i wyświetla małą „strzałkę", wskazującą na najwcześniejszy punkt w wierszu, w którym błąd został znaleziony.

Aby zilustrować typowy błąd parsingu, rozważmy przykład użycia pętli `while` z celowo popełnionym błędem.

```
>>> while True print("Hello")
File "<stdin>", line 1
while True print("Hello")
           ^
SyntaxError: invalid syntax
```

W powyższym przykładzie błąd został wykryty w składniku poprzedzającym strzałkę. Interpreter Pythona wykrył go w instrukcji wywołania funkcji `print`, z powodu braku znaku dwukropka (":") przed tą instrukcją. Ponadto w komunikacie o błędzie wyświetlane są również: numer linii, aby wiedzieć, gdzie szukać błędu oraz nazwa `<stdin>` - będąca standardowym wejściem, gdy pracujemy z interpreterem lub nazwą pliku - w przypadku gdy na wejściu znajdował się skrypt.

6.2. Obsługa wyjątków

Pomimo dbania o poprawność składniową wyrażenia, może się zdarzyć, że pojawią się błędy podczas próby jego wykonania. O błędach wykrywanych podczas wykonania mówimy, że są wyjątkami (ang. *exceptions*), co ważne, niekoniecznie muszą doprowadzić do zakończenia programu.

Większość wyjątków nie jest obsługiwana przez programy, a o ich wystąpieniu informują nas komunikaty o błędzie. W poniższych przykładach przedstawiamy najczęściej występujące błędy podczas wykonania programów: próba dzielenia przez zero, odwołanie się do niezdefiniowanej zmiennej czy próba sklejania łańcucha z liczbą.

```
>>> 10 * 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> 4 * 3 + spam
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>>
>>> "2" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Wyświetlany napis informujący o typie wyjątku, który się właśnie pojawił, jest jego wbudowaną nazwą. Ponadto w komunikacie w szczegółach opisane jest to, co się stało, a interpretacja i znaczenie tego opisu zależy od typu wyjątku, który wystąpił.

Do najważniejszych grup błędów występujących w Pythonie należą:

- `Syntax error` - zła składnia, interpreter kończy działanie;
- `Type error` - nie można wykonać operacji w ramach danego typu, np. dodać wartości liczbowej do napisu;
- `Index error` - przekroczenie zasięgu listy lub krotki;
- `Value error` - nieobsługiwana wartość lub typ danych;
- `Zero division error` - błąd dzielenia przez zero.

Jeżeli wiemy, że w danym miejscu kodu może dojść do wyjątku i zatrzymania programu, powinniśmy próbować taką sytuację obsłużyć przy pomocy specjalnego bloku: `try...except...else` lub `try...except...else...finally`.

W pierwszej kolejności, zgodnej z ich zapisem, wykonywane są instrukcje w klauzuli `try` (czyli instrukcje pomiędzy `try` a `except`). Gdy nie pojawi się żaden wyjątek, klauzula `except` jest pomijana, a wykonanie instrukcji `try` uważa się za zakończone. Jeżeli podczas wykonywania klauzuli `try` wystąpi wyjątek, pozostałe, niewykonane jeszcze instrukcje są pomijane. Następnie, w zależności od tego, czy typ wyjątku pasuje do jednego z typów wyjątków wymienionych w klauzuli/klauzulach `except`, wykonywany jest kod zawarty w dopasowanej klauzuli i interpreter przechodzi do wykonywania instrukcji umieszczonych po całej instrukcji `try...except`. W przypadku pojawienia się wyjątku, który nie zostanie dopasowany do żadnego z wyjątków wymienionych w klauzuli/klauzulach `except`, zostaje on przekazany do następnych, zewnętrznych instrukcji `try`. Jeżeli również tam nie zostanie znaleziona odpowiadająca mu klauzula `except`, wyjątek ten nie zostanie zauważony: stanie się on nieobsłużonym wyjątkiem, a wykonywanie programu zostanie wstrzymane wraz z pojawieniem się odpowiedniego komunikatu. Aby umożliwić obsługę wielu wyjątków instrukcja `try` może posiadać więcej niż jedną klauzulę `except` lub pojedyncza klauzula `except` może zawierać nazwy wielu wyjątków, podanych w formie krotki otoczonej nawiasami okrągłymi, np.

```
except (TypeError, NameError): pass
```

Jako ostatnia może wystąpić klauzula `except` z pominiętą nazwą/nazwami wyjątków, która obsługiwać będzie dowolny wyjątek. Na listingu 6.1 w liniach 15 - 18 użyliśmy tego typu klauzuli, aby wypisać komunikat o błędzie i ponownie zgłosić wychwycony wyjątek za pomocą instrukcji `raise`, umożliwiając w ten sposób funkcji wywołującej, w przypadku programu z listingu 6.1 - funkcji `main`, wychwycenie zgłoszonego wyjątku.

Zwracamy szczególną uwagę na posługiwanie się konstrukcją `except` z pominiętą nazwą wyjątku, gdyż nieostrożne jej użycie może spowodować zamaskowanie prawdziwego błędu występującego w programie!

Instrukcja `try ... except` wyposażona jest również w opcjonalną klauzulę `else`, która występuje za wszystkimi podanymi blokami `except`. Można w niej umieścić kod, który zostanie wykonany, o ile żaden wyjątek nie zostanie zgłoszony. Umieszczenie kodu w klauzuli `else` zamiast poza całymi klauzulami `try-except` jest o tyle bezpieczniejsze, że można uniknąć przypadkowego wystąpienia wyjątku, który nie został wcześniej zgłoszony przez kod chroniony w klauzuli `try`.

Listing 6.1: *Obsługa dzielenia przez zero lub każdego innego wyjątku*

```
1 def demo1(x):
2   try:
3     print(1/x)
4   except ZeroDivisionError:
5     print("x =", x, ":Nie można dzielić przez zero")
```

```
6     except:
7       print("x =", x, ":Nieoczekiwany błąd")
8       raise
9     else:
10      print("Udało się :-)")
11
12  if __name__ == "__main__":
13    a = float(input("Podaj liczbę: "))
14    demo1(a)
15    try:
16      demo1(str(a))
17    except:
18      print("Zgłoszono ponownie wyjątek")
```

Na listingu 6.1 przedstawiamy obsługę wyjątku dzielenia przez zero (linia 4) z uwzględnieniem również i innych sytuacji wyjątkowych (linia 6). Możliwe wywołania powodujące wystąpienie wyjątków mogą wyglądać w następujący sposób:

- po podaniu przez użytkownika wartości `0` lub `0.0`:

```
Podaj liczbę: 0
x = 0.0 : Nie można dzielić przez zero
Ponowne wywołanie funkcji <demo1>
dla argumentu typu <str>
x = 0.0 : Nieoczekiwany błąd
Zgłoszono ponownie wyjątek
```

- po podaniu przez użytkownika wartości różnej od zera:

```
Podaj liczbę: 12.0
0.08333333333333333
Udało się :-)
Ponowne wywołanie funkcji <demo1>
dla argumentu typu <str>
x = 12.0 : Nieoczekiwany błąd
Zgłoszono ponownie wyjątek
```

Za każdym razem ponowne wywołanie funkcji `demo1` w bloku `try` w funkcji `main` (linie 15-18) powoduje, że w ciele funkcji `demo1` wykonywany jest kod zawarty w bloku `except` (linie 6-8) i następuje ponowne zgłoszenie wyjątku, który obsługiwany jest już w funkcji wywołującej. Stąd w wyniku wywołania programu z listingu 6.1 w obydwu wyżej wymienionych przykładach dwie ostatnie linie komunikatu na ekranie są podobnej treści.

W bloku `try` ostatnią (opcjonalną) klauzulą jest `finally`, której kod zostanie uruchomiony zawsze na zakończenie obsługi wyjątków - niezależnie od tego, czy w programie pojawią się wyjątki czy nie. Na listingu 6.2 przedstawiamy użycie tej klauzuli, która na zakończenie obsługi wyjątków zawsze wypisze na ekran informację o końcu bloku obsługi wyjątków (linie 9-10). W klauzuli `except` można użyć frazy `as` i nazwy zmiennej, np. `er` jak uczyniliśmy to w linii 4. Wówczas do zmiennej `er` przypisywany jest obiekt wyjątku, dzięki temu możemy uzyskać dodatkowe informacje o zaistniałej wyjątkowej sytuacji i na przykład wyświetlić je na ekran (linia 6).

LISTING 6.2: *Użycie klauzuli* `finally`

```
1  def demo2(x):
2    try:
3      print(1/x)
4    except ZeroDivisionError as er:
5      print("x =", x, ":Nie można dzielić przez zero")
6      print("Exception",type(er).__name__,"(",er,") occurred")
7    else:
8      print("Wyjątków brak")
9    finally:
10     print("Zawsze na koniec try!")
11
12 if __name__ == "__main__":
13   a = float(input("Podaj liczbę: "))
14   demo2(a)
```

Na listingu 6.3 prezentujemy w linii 4 zgłoszenie wyjątku przez funkcję, która jest obiektem klasy `ValueError` wraz z komunikatem o błędzie, który w tym przypadku jest argumentem typu string tworzonego obiektu. W funkcji `main` w klauzuli `except`, obsługującej ten typ wyjątku, związana jest z nim zmienna `e`, dzięki czemu na ekran wypisany zostanie sformułowany wcześniej komunikat o błędzie.

LISTING 6.3: *Zgłaszanie wyjątku*

```
1  def demo3():
2    x = int(input("Podaj liczbę naturalną:   "))
3    if x < 0: raise ValueError("Liczba *nie może* być ujemna")
4    else: return x
5
6  if __name__=='__main__':
7    try:
8      print(demo3())
9    except  ValueError as e:
10     print(e)
```

Na listingach 6.4 oraz 6.5 zdefiniowaliśmy dwie funkcje do konwersji łańcuchów do wartości typu `int` oraz `float` odpowiednio, obsługujące sytuacje wyjątkowe z nią związane.

LISTING 6.4: *Obsługa konwersji na typ* `int`

```
1  def inputInt(prompt):
2    while True:
3      try:
4        a = int(input(prompt))
5      except:
6        print("To nie jest liczba całkowita")
7        print("Wpisz ponownie, aż do skutku!")
8      else:
9        return a
10 if __name__=='__main__':
11   print(inputInt("Podaj wartość całkowitą:"))
```

LISTING 6.5: *Obsługa konwersji na typ* `float`

```
1  def inputFloat(prompt):
2    while True:
3      try:
4        a = float(input(prompt))
5      except:
6        print("To nie jest liczba typu float")
7        print("Wpisz ponownie, aż do skutku!")
8      else:
9        return a
10 if __name__=='__main__':
11   print(inputFloat("Podaj wartość rzeczywistą:"))
```

Jak już wspomnieliśmy wcześniej, klauzula `raise` służy do samodzielnego zgłaszania błędu, jeżeli zajdzie jakaś sytuacja w kodzie, o której istnieniu chcemy poinformować użytkownika - bardziej innego programistę, który będzie z naszego kodu korzystał, niż użytkownika końcowego (linia 2 z listingu 6.6). Zmienna `inst` z linii 3 z listingu 6.6 jest powiązana z instancją wyjątku `Exception`, co potwierdza typ zmiennej (linia 4). Instancja ta posiada atrybut `args` przechowujący w krotce argumenty (linia 5). Typy wyjątków wbudowanych definiują również metodę `__str__()`, która wypisuje wszystkie argumenty bez potrzeby odwoływania się do atrybutu `args` (linia 6). Zdecydowanie nie polecamy obsługi wyjątków poprzez obsługę bazowego `Exception`. W sytuacji wyjątkowej należy jak najdokładniej dopasować odpowiedni wbudowany typ wyjątku lub zdefiniować własną klasę do jego obsługi.

Listing 6.6: *Zgłaszanie wyjątku* `Exception`

```
try:
  raise Exception('jeden', 'dwa')
except Exception as inst:
  print(type(inst))
  print(inst.args)
  print(inst)
  x, y = inst.args # rozpakowanie krotki inst.args
  print('x =', x)
  print('y =', y)
```

Z kolei na listingu 6.7 prezentujemy obsługę sytuacji wyjątkowej związanej z wystąpieniem błędu w linii 3 przy próbie konwersji liczby z zapisu szesnastkowego na dziesiętny, gdzie w zapisie liczby `"100"` zamiast zera została użyta wielka litera `'O'`.

Listing 6.7: *Obsługa konwersji z zapisu szesnastkowego do dziesiętnego*

```
def main():
  print(str_to_int("ABCD"))
  print(str_to_int("10O"))

def str_to_int(string):
  try:
    return int(string, 16)
  except ValueError as e:
    print("Błąd  wartości:", e)

if __name__ == "__main__":
  main()
```

Na listingu 6.8 wykorzystamy obsługę wyjątków i argumentów wiersza poleceń do obliczenia najmniejszej sumy liczb `1 + 1/2 + 1/3 ...` większej od wartości przekazanego argumentu, który powinien być liczbą rzeczywistą.

Listing 6.8: *Suma odwrotności liczb naturalnych*

```
from sys import argv

def main():
  if len(argv) != 2: a = 2.0
  else:
    try: a = float(argv[1])
    except: a = 3.0
  print(suma_odwrotnosci(a))
```

```
9  def suma_odwrotnosci(a):
10   n, s = 0, 0.0
11   while s <= a:
12     n += 1
13     s += 1 / n
14   return s
15
16 if __name__ == "__main__":
17   main()
```

W funkcji `main` w linii 4 testowany jest warunek poprawności wywołania programu z listingu 6.8. Jeżeli program został wywołany dla zera lub więcej niż jednego argumentu, to jako ograniczenie dla obliczanej sumy odwrotności kolejnych liczb naturalnych przyjmujemy wartość `2.0`. W przeciwnym przypadku, w bloku `try` (linia 6) odczytujemy i konwertujemy na typ `float` wartość argumentu `sys.argv[1]`. Jeśli nie jest on liczbą, konwersja nie powiedzie się i zostanie zgłoszony wyjątek, który następnie będzie obsłużony w bloku `except` (linia 7) i jako ograniczenie zostanie przyjęta wartość `3.0`. Dzięki tym zabiegom wywołanie funkcji `suma_odwrotnosci` zawsze zwróci i wypisze na ekran wynik dokonanych w jej ciele obliczeń.

```
~$ python3 main_arg_4.py 6.0
6.004366708345567
~$ python3 main_arg_4.py
2.083333333333333
~$ python3 main_arg_4.py 6.0 4
2.083333333333333
~$ python3 main_arg_4.py "Ala"
3.0198773448773446
```

Na listingu 6.9 przedstawiamy przykład programu obsługującego wyjątek przy pomocy pełnej instrukcji `try-except-else-finally`, z użyciem instrukcji `raise` (linia 6) oraz klasy błędu zdefiniowanego przez tworzącego program (linia 1).

Listing 6.9: *Zgłaszanie wyjątku z jego obsługą przy pomocy własnej klasy błędu*

```
1 class NiedodatniaError(Exception): pass
2
3 try:
4   a = float(input('Podaj bok kwadratu:'))
5   if a <= 0:
6     raise NiedodatniaError
7 except ValueError:
8   print('Zły format')
```

```
except NiedodatniaError:
  print('Liczba miała być dodatnia')
else:
  print('Pole kwadratu', a**2)
finally:
  print('Koniec programu.')
```

6.3. Obsługa plików

Z punktu widzenia systemu operacyjnego, plik jest ciągiem bajtów o określonej długości zapisanym na trwałym nośniku informacji. W `Pythonie` rozróżnia się pliki tekstowe oraz pliki binarne. Plik tekstowy to ciąg znaków domyślnie zapisanych z użyciem kodowania `UTF-8`, a plik binarny to ciąg bajtów. `UTF-8` to system kodowania `Unicode`, wykorzystujący od 1 do 4 bajtów do zakodowania pojedynczego znaku, w pełni zgodny z `ASCII`. Zgodność kodowania `UTF-8` z `ASCII` oznacza, że każdy łańcuch znaków `ASCII` jest także ciągiem znaków w `UTF-8`. W `Pythonie` pliki są też nazywane `strumieniami`. Dostęp do plików uzyskuje się w `Pythonie` poprzez standardową (dostępną bez importu dodatkowych modułów) klasę `file`. Utworzenie obiektu tej klasy wiąże się z otwarciem pliku pozwalającym na zapis i/lub odczyt danych. Funkcja `open` zwraca się do systemu operacyjnego, aby zezwolił na dostęp (odczyt/zapis) do pliku. Gdy to się uda, zwracany jest obiekt klasy `file`, który posiada metody i atrybuty pozwalające wykonywać na pliku pewne operacje.

Najczęściej używaną postacią wywołania funkcji `open` jest ta z dwoma argumentami: `open(nazwa_pliku, tryb)`. Pierwszym argumentem jest łańcuch znaków określający nazwę pliku. Drugim argumentem jest napis opisujący tryb otwarcia pliku. Argument `tryb` jest opcjonalny: w przypadku jego braku, plik zostanie otwarty w formacie tekstowym w trybie tylko do odczytu. Istotne atrybuty obiektu pliku to: `name` (nazwa pliku), `mode` (tryb otwarcia pliku), `encoding` (kodowanie, istnieje tylko dla plików tekstowych) oraz `closed` („Czy zamknięty?" Zwraca `True`, gdy plik został zamknięty, w przeciwnym przypadku zwraca `False`). Wywołanie dla obiektu pliku metody `close` zamyka plik i natychmiast zwalnia wszystkie zasoby systemowe wykorzystywane przez ten plik.

```
>>> f = open('/etc/passwd')
>>> f.name
'/etc/passwd'
>>> f.mode
'r'
>>> f.encoding
'UTF-8'
```

```
>>> f.closed
False
>>> f.close()
>>> f.closed
True
```

6.3.1. Tryby otwarcia plików

`Tryby otwarcia pliku do odczytu`:

- W każdym z przypadków z tabeli 6.1, po otwarciu pliku wskaźnik pliku jest umieszczany na jego początku.
- Jeżeli plik o podanej nazwie nie istnieje, zgłaszany jest wyjątek: `FileNotFoundError`.

Tabela 6.1: Tryby otwarcia pliku do odczytu

Tryb	Opis
`r`	Otwiera plik w formacie tekstowym tylko do odczytu (jest to tryb domyślny).
`rb`	Otwiera plik w formacie binarnym tylko do odczytu.
`r+`	Otwiera plik w formacie tekstowym do odczytu i zapisu.
`rb+`	Otwiera plik w formacie binarnym do odczytu i zapisu.

`Tryby otwarcia pliku do zapisu`:

- W każdym z przypadków z tabeli 6.2, jeżeli plik o podanej nazwie istnieje, to jego zawartość jest nadpisywana.
- Jeżeli plik o podanej nazwie nie istnieje, tworzony jest nowy plik.

Tabela 6.2: Tryby otwarcia pliku do zapisu

Tryb	Opis
`w`	Otwiera plik w formacie tekstowym tylko do zapisu.
`wb`	Otwiera plik w formacie binarnym tylko do zapisu.
`w+`	Otwiera plik w formacie tekstowym do odczytu i zapisu.
`wb+`	Otwiera plik w formacie binarnym do odczytu i zapisu.

`Tryby otwarcia pliku do dopisywania`:

- W każdym z przypadków z tabeli 6.3, po otwarciu pliku wskaźnik pliku jest umieszczany na jego końcu.
- Jeżeli plik o podanej nazwie nie istnieje, tworzony jest nowy plik.
- Z kolei w każdym z przypadków z tabeli 6.4, jeżeli plik o podanej nazwie istnieje, zgłaszany jest wyjątek: `FileExistsError`.

Tabela 6.3: Tryby otwarcia pliku do dopisywania

Tryb	Opis
a	Otwiera plik w formacie tekstowym tylko do dopisywania.
ab	Otwiera plik w formacie binarnym tylko do dopisywania.
a+	Otwiera plik w formacie tekstowym do odczytu i dopisywania.
ab+	Otwiera plik w formacie binarnym do odczytu i dopisywania.

Tabela 6.4: Tryby otwarcia pliku do dopisywania

Tryb	Opis
x	Otwiera plik w formacie tekstowym tylko do zapisu.
xb	Otwiera plik w formacie binarnym tylko do zapisu.
x+	Otwiera plik w formacie tekstowym do odczytu i zapisu.
xb+	Otwiera plik w formacie binarnym do odczytu i zapisu.

Zazwyczaj pliki są otwierane w trybie tekstowym (ang. *text mode*), co oznacza, że czytamy i zapisujemy ciągi znaków do i z pliku. Ciągi, które są kodowane zgodnie z podanym opcjonalnym, trzecim argumentem `encoding`. Kodowanie domyślne jest zależne od platformy. Ponieważ `UTF-8` jest współczesnym standardem kodowania, zaleca się stosowanie `encoding="utf-8"`, chyba że wiemy, że musimy użyć innego. W trybie binarnym (ang. *binary mode*) dane są odczytywane i zapisywane jako obiekty `bytes`. Dla tego typu plików podczas ich otwierania nie można określać kodowania.

6.3.2. Wybrane metody klasy `file`

Zakładamy, że obiekt pliku `f` został już utworzony i skojarzony z fizycznym plikiem na dysku o nazwie `plik1.txt` i o zawartości:

```
Pierwszy wiersz.
Drugi wiersz.
Trzeci wiersz.
Ostatni, następny wiersz jest pusty.
```

Nie będziemy w tym miejscu dbać o tryb otwarcia pliku, zilustrujemy tylko sposób użycia metod obiektu `f`.

- `f.readable()` – zwraca `True`, jeżeli z pliku można czytać, a `False` w przeciwnym przypadku, np.

  ```
  >>> f.readable()
  True
  ```

- `f.writable()` – zwraca `True`, jeżeli do pliku można zapisywać, a `False` w przeciwnym przypadku, np.

  ```
  >>> f.writable()
  False
  ```

- `f.close()` - zamyka plik oraz zwalnia wszystkie zasoby systemowe związane z otwarciem i obsługą tego pliku. Po wywołaniu `f.close()` każda próba użycia metody obiektu pliku `f` spowoduje wygenerowanie błędu:
  `ValueError: I/O operation on closed file`.
  W przypadku niezamknięcia pliku przez program, metoda `close` zostanie wywołana przez interpreter przy ostatecznym niszczeniu obiektu. Przykładowo:

  ```
  >>> f.close()
  >>> f.readable()
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: I/O operation on closed file
  ```

- `f.read(size)` - odczytuje zawartość pliku odczytując pewną ilość danych (`size`) i zwraca je jako ciąg znaków (w trybie tekstowym) lub obiekt bajtowy (w trybie binarnym), `size` jest opcjonalnym argumentem numerycznym i gdy go pominiemy lub jest wartością ujemną, zostanie odczytana i zwrócona cała zawartość pliku. W przeciwnym razie odczytanych i zwróconych zostanie co najwyżej `size` znaków (w trybie tekstowym) lub `size` bajtów (w trybie binarnym). Jeśli został osiągnięty koniec pliku, `f.read()` zwróci pusty ciąg znaków (`''`). Przykładowo:

  ```
  >>> f.read()
  'Pierwszy wiersz.\nDrugi wiersz.\nTrzeci wiersz.\n
  Ostatni, następny wiersz jest pusty.\n\n'
  ...
  >>> f.read(12)
  'Pierwszy wie'
  ...
  >>> f.read(-1)
  'Pierwszy wiersz.\nDrugi wiersz.\nTrzeci wiersz.\n
  Ostatni, następny wiersz jest pusty.\n\n'
  ```

- `f.readline()` - zwraca napis, którego zawartością jest przeczytany z pliku pojedynczy wiersz. Znak nowej linii (`\n`) jest pozostawiony na końcu ciągu znaków i jest pomijany tylko w ostatniej linii pliku, jeśli plik nie kończy się nową linią. Po przeczytaniu wszystkich wierszy pliku, kolejne wywołanie zwraca pusty napis (`''`). Pusta linia jest reprezentowana przez `'\n'`. Przykładowo:

```
>>> f.readline()
'Pierwszy wiersz.\n'
>>> f.readline()
'Drugi wiersz.\n'
>>> f.readline()
'Trzeci wiersz.\n'
>>> f.readline()
'Ostatni, następny wiersz jest pusty.\n'
>>> f.readline()
'\n'
>>> f.readline()
''
```

- `f.readlines()` - zwraca listę napisów, którymi są kolejne wiersze pliku, od pierwszego do ostatniego. Innym sposobem odczytania wierszy z pliku jest wykonanie pętli na obiekcie pliku, co jest wydajne pamięciowo i szybkie, a dodatkowo dość zrozumiałe, jeśli weźmiemy pod uwagę, że plik jest domyślnie strukturą sekwencyjną iterowalną (elementami iteracji są wiersze pliku), np.

```
>>> for line in f: print(line, end='')
...
Pierwszy wiersz.
Drugi wiersz.
Trzeci wiersz.
Ostatni, następny wiersz jest pusty.


>>>
```

  Jeszcze innym sposobem wczytania wszystkich wierszy z pliku jest użycie funkcji `list` do utworzenia z nich listy, np.

```
>>> list(f)
['Pierwszy wiersz.\n', 'Drugi wiersz.\n', 'Trzeci wiersz.\n',
'Ostatni, następny wiersz jest pusty.\n', '\n']
```

- `f.write(string)` – zapisuje zawartość napisu `string` do pliku. Zwraca liczbę znaków zapisanych do pliku. Przykładowo:

```
>>> f.write('To jest kolejny wiersz.\n')
24
```

Typy niebędące napisem muszą zostać przekonwertowane, odpowiednio na ciąg znaków (w trybie tekstowym) albo na obiekt bajtowy (w trybie binarnym), przed ich zapisem do pliku, np.:

```
>>> liczba = 123
>>> f.write('To jest liczba całkowita' + str(liczba) + '\n')
28
```

- `f.writelines(lines)` – zapisuje listę napisów `lines` do pliku. Separatory wierszy nie są dodawane, więc należy pamiętać, aby samodzielnie dopisać separator na końcu każdego wiersza. Przykładowo:

  ```
  >>> napisy = ['Pierwszy\n', 'Drugi\n','Trzeci\n']
  >>> f.writelines(napisy)
  ```

- `f.flush()` - zapisuje dane z bufora do pliku bez jego zamykania. UWAGA: na niektórych obiektach plikowych może być pustą operacją.

6.3.3. Przykłady pracy z plikami tekstowymi

Na listingu 6.10 przedstawiamy program, w którym tworzymy plik z kodowaniem znaków `encoding = 'UTF-16'` (linie 1-3), a następnie próbujemy czytać z pliku otwartego z domyślną wartością argumentu `encoding = 'UTF-8'` (linie 4-6). Niezgodność kodowania powoduje zgłoszenie wyjątku:

```
UnicodeDecodeError: 'utf-8' codec can't decode
byte 0xff in position 0: invalid start byte
```

Listing 6.10: *Kodowanie znaków w plikach*

```
1 f = open("utf16.txt", "w", encoding="UTF-16")
2 f.write("Ala ma kota")
3 f.close()
4 f = open("utf16.txt")
5 content = f.read()
6 f.close()
```

W kolejnym przykładzie przedstawionym na listingu 6.11 tworzymy zmienną `wierszyk`, która jest wielowierszowym ciągiem znaków (linie 3-8). Bezpośrednio po otwarciu tego napisu jest użyty znak kontynuacji '\'. Gdyby go nie było, napis zawierałby dodatkową pustą linijkę na samym początku. Otwieramy plik o nazwie `'wierszyk.txt'` do zapisu (linia 7) i zapisujemy do niego tekst podzielony na wiersze (linia 8). Należy pamiętać, że zapis danych na dysk nie następuje natychmiast, ale dopiero wtedy, gdy uzbiera

się ich większa ilość. Stąd chcąc kontynuować pracę na pliku, ale w trybie do odczytu, powinno się wykonać operację odwrotną do otwarcia, czyli `close` (linia 9). Wówczas dane zostaną zapisane na dysk i obiekt pliku zostanie skasowany. Jeśli jednak chcemy zacząć odczytywać ten plik bez jego wcześniejszego zamykania - musimy się upewnić, że nasze zmiany naprawdę znalazły się na dysku i tu przydatna okazuje się metoda `flush` (komentarz w linii 9). Po uaktualnieniu pliku `'wierszyk.txt'`, ponownie zostaje on otwarty, ale tym razem do odczytu (linia 10), a jego zawartość odczytywana jest wierszami i wypisywana na ekran (linie 11-12).

LISTING 6.11: *Przykład kilkukrotnego otwarcia pliku*

```
1  wierszyk = '''\
2  Eeny, meeny, miny, moe,
3  Catch a tiger by the toe.
4  If he hollers, let him go,
5  Eeny, meeny, miny, moe.
6  '''
7  f1 = open('wierszyk.txt', 'w')
8  f1.write(wierszyk)
9  f1.close() #; f1.flush()
10 f2 = open('wierszyk.txt')
11 for wers in f2:
12     print(wers)
```

Zamykanie plików często jest pomijane (jak ma to miejsce na końcu kodu w przykładzie z listingu 6.11), bo wiemy, że Python i tak z chwilą zakończenia programu wszystko za nas zamknie i zwolni zasoby związane z obiektem pliku. Aby zapobiec błędom wynikającym z niezamknięcia pliku, gdy jest to wymagane, wprowadzono w Pythonie instrukcję `with` i wyposażono obiekty otwierane (takie jak `file`) w funkcjonalność pozwalającą na automatyczne ich zamykanie, bowiem obiekt jest automatycznie zamykany z chwilą zakończenia bloku `with`. Na listingu 6.12 zilustrowano prawidłowe użycie tej instrukcji.

LISTING 6.12: *Użycie instrukcji* `with`

```
1 with open('wierszyk.txt') as plik:
2     for wers in plik:
3         print(wers, end='')
```

Na listingu 6.13 przedstawiamy kod programu odczytującego kolejne linie z pliku tekstowego przy użyciu instrukcji `while` (linie 4-10) oraz metody `readline` (linia 4). Odczytanie ostatniej linii z pliku, która jest pusta (linie 5-6), powoduje wyjście z pętli, zamknięcie pliku (linia 8) i zakończenie programu.

Listing 6.13: *Użycie pętli* `while` *do odczytu pliku*

```
1  def main():
2    f = open("wierszyk.txt", "r")
3    while True:
4      linia = f.readline()
5      if len(linia) == 0:
6        break
7      print(linia, end = "")
8    f.close()
9
10 if __name__ == '__main__':
11   main()
```

Na listingu 6.14 przedstawiamy kod programu tworzącego posortowaną kopię pliku tekstowego poprzez zapis wszystkich linii na raz. Dzięki odczytaniu całej zawartości pliku, jako listy jego wierszy (linia 3), mamy możliwość wykorzystania dla listy metody sortującej w porządku rosnącym jej zawartość (linia 5), którą następnie zapisujemy do pliku wyjściowego (linia 7).

Listing 6.14: *Tworzenie posortowanej wierszami kopii pliku*

```
1  def main():
2    f = open("wierszyk.txt", "r")
3    lines = f.readlines()
4    f.close()
5    lines.sort()
6    g = open("posortowanywierszyk.txt", "w")
7    g.writelines(lines)
8    g.close()
9
10 if __name__ == '__main__':
11   main()
```

W programie z listingu 6.15 użytkownik podaje nazwę pliku (linia 2). Jeżeli nazwa nie zostanie podana, program wypisuje komunikat na ekranie i kończy działanie (linie 3-4). W przeciwnym przypadku, próbuje otworzyć plik do odczytu w bloku `try` (linia 7) i jeśli operacja nie powiedzie się, program kończy działanie po wypisaniu odpowiedniego komunikatu na ekranie (linia 9). Jeżeli z powodzeniem uda się otworzyć plik, wykonywany jest kod w klauzuli `else` (linie 11-16), który tworzy słownik znaków i liczby ich wystąpień w pliku, a następnie wypisuje go na ekranie.

LISTING 6.15: *Zwraca słownik znaków i liczbę ich wystąpień w pliku*

```
def main():
  nazwa = input("Podaj nazwe pliku: ")
  if len(nazwa) == 0:
    print('Nie wprowadzono nazwy pliku!')
  else:
    try:
      f1 = open(nazwa, "r")
    except:
      print("Nie udało się otworzyć pliku")
    else:
      D = {}
      for linie in f1:
        for znak in linie:
          D[znak.lower()] = D.get(znak.lower(),0) + 1
      f1.close()
      print(D)

if __name__ == '__main__':
  main()
```

Sugerujemy korzystanie z obsługi wyjątków przy pracy z plikami, gdzie szczególnie w przypadku próby otwarcia pliku, operacja ta może zakończyć się niepowodzeniem. Warto wówczas poinformować użytkownika, z jakiego powodu program został zakończony lub odpowiednio obsłużyć zaistniałą sytuację.

6.4. Pliki binarne

Zanim przejdziemy do zaprezentowania działań na plikach binarnych, omówimy binarne typy sekwencyjne - klasy `bytes` oraz `bytearray`.

Obiekty klasy `bytes` są niezmiennymi sekwencjami pojedynczych bajtów. Każdy bajt ma wartość całkowitą z przedziału od 0 do 255. Większość protokołów binarnych bazuje na kodowaniu tekstowym `ASCII`, dlatego klasa `bytes` oferuje metody, które działają tylko z danymi kompatybilnymi z `ASCII`. W literałach bajtów dopuszczalne są tylko znaki `ASCII` (niezależnie od zadeklarowanego kodowania kodu źródłowego).

Funkcja `bytes([source[,encoding[,errors]]])` pozwala utworzyć obiekt klasy `bytes`, gdzie:

- `source`: źródło. Jeśli `source` jest ciągiem znaków (`str`), wymagane jest podanie parametru `encoding` (i opcjonalnie `errors`); następnie łańcuch konwertowany jest na bajty za pomocą `str.encode()`. Jeśli `source` jest liczbą całkowitą (`int`), sekwencja

będzie miała taki rozmiar, jak ta liczba i zostanie zainicjowana bajtami zerowymi. Jeśli `source` jest obiektem iterowalnym, to musi to być obiekt liczb całkowitych z zakresu od 0 do 255, które są używane jako początkowa zawartość sekwencji. Bez argumentu metoda `bytes()` utworzy sekwencję o rozmiarze 0.

- `encoding`: parametr opcjonalny. Wymagane, jeśli `source` jest ciągiem znaków. Przykład: `'utf-8'`, `'ascii'` itp.
- `errors`: parametr opcjonalny dla `source` będącego ciągiem znaków. W zależności od potrzeby może mieć wartości takie jak:`'strict'` - zgłaszanie wyjątków, `'replace'` - zastępowanie przez `'?'`, `'ignore'` - pomijanie błędnego znaku itp.

Tworzenie pustego obiektu klasy `bytes`:

```
>>> x = bytes()
>>> print(x)
b' '
```

Tworzenie sekwencji o długości 3 bajtów wypełnionej zerowymi bajtami:

```
>>> x = bytes(3)
>>> print(x)
b'\x00\x00\x00'
```

Tworzenie sekwencji o długości listy wypełnionej jej wartościami:

```
>>> x = bytes([1,2,3])
>>> print(x)
b'\x01\x02\x03'
```

Tworzenie sekwencji znaków w kodzie `'UTF-8'`:

```
>>> x = bytes('Python','utf-8')
>>> print(x)
b'Python'
```

Tworzenie sekwencji znaków w kodzie `'ASCII'`:

```
>>> x = bytes('Python', 'ascii')
>>> print(x)
b'Python'
```

Próba utworzenia obiektu klasy `bytes` dla źródła będącego napisem bez podania sposobu kodowania kończy się niepowodzeniem i zgłoszeniem błędu `TypeError`:

```
>>> bytes('źrebak')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding
```

Podanie sposobu kodowania, gdzie w źródle znajdują się znaki spoza tego kodu, również kończy się wskazaniem przez interpreter błędu, ale tym razem błędu kodowania `UnicodeEncodeError`:

```
>>> bytes('źrebak','ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character
'\u017a' in position 0: ordinal not in range(128)
```

Podobny komunikat uzyskamy, gdy zastosujemy trzeci argument funkcji `bytes` o wartości `'strict'`:

```
>>> bytes('źrebak','ascii','strict')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character
'\u017a' in position 0: ordinal not in range(128)
```

Użycie opcji `'ignore'` jako trzeciego argumentu tworzonego obiektu `bytes` spowoduje zignorowanie każdego znaku nienależącego do wskazanego zbioru kodów i utworzenie obiektu z pozostałych, prawidłowych znaków:

```
>>> bytes('źrebak','ascii','ignore')
b'rebak'
```

W przypadku, gdy zastosujemy opcję `'replace'`, każdy znak spoza zbioru kodowego zostanie zastąpiony w tworzonym obiekcie `bytes` znakiem zapytania `'?'`:

```
>>> bytes('źrebak','ascii','replace')
b'?rebak'
```

Na koniec poprawne utworzenie obiektu `bytes` uwzględniającego wszystkie występujące w źródle znaki:

```
>>> bytes('źrebak','utf-8')
b'\xc5\xbarebak'
```

Nie utworzymy również obiektu `bytes`, jeżeli wartości wskazanego jako źródła przedziału liczbowego wykraczają poza dopuszczalne wartości:

```
>>> bytes(range(100,300))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)
```

Również wtedy, gdy źródłem jest lista wartości nienależąca do wymaganego zbioru:

```
>>> bytes([1,2,300])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)
```

I jeszcze raz utworzenie obiektu `bytes` składającego się z wybranych polskich znaków diakrytycznych:

```
>>> bytes("ąćęłńó", "utf-8")
b'\xc4\x85\xc4\x87\xc4\x99\xc5\x82\xc5\x84\xc3\xb3'
```

który następnie przekształcany jest w listę liczb kodowych tych znaków:

```
>>> list(bytes("ąćęłńó", "utf-8"))
[196, 133, 196, 135, 196, 153, 197, 130, 195, 179]
```

Na listingu 6.16 przedstawiamy kod programu przekształcającego łańcuchy znaków w sekwencje bajtów. Na początku tworzymy dwie zmienne (linie 1-2): `a`, która jest napisem i `c`, która jest łańcuchem bajtów. Następnie kodujemy łańcuch `a` na typ `bytes` (linia 4) i sprawdzamy, czy obiekt `d` jest typu `bytes` (linie 6-9). Porównując obiekt `d` z obiektem referencyjnym `c` dowiadujemy się, czy kodowanie zakończyło się powodzeniem (linie 11-14).

LISTING 6.16: *Przekształcanie* `str` *do* `bytes`

```
1  a = 'Podstawy programowania w Pythonie' # <class 'str'>
2  c = b'Podstawy programowania w Pythonie' # <class 'bytes'>
3
4  d = a.encode('ASCII')
5
6  if isinstance(d,bytes):
7    print(type(d))
8  else:
9    print(type(a))
10
11 if (d==c):
12   print ("Kodowanie zakończyło się powodzeniem")
13 else:
14   print ("Kodowanie nie powiodło się")
```

Na listingu 6.17 przedstawiamy kod programu przekształcającego sekwencje bajtów w łańcuchy znaków. Na początku tworzymy dwie zmienne (linie 1-2): `a`, która jest łańcuchem znaków i `c`, która jest łańcuchem bajtów. Następnie dekodujemy sekwencję bajtów

`c` na typ `str` (linia 4) i sprawdzamy, czy obiekt `d` jest typu `str` (linie 6-9). Porównując obiekt `d` z obiektem referencyjnym `a` dowiadujemy się, czy kodowanie zakończyło się powodzeniem (linie 11-14).

Listing 6.17: *Przekształcanie* `bytes` *do* `str`

```
1  a = 'Podstawy programowania w Pythonie' # <class 'str'>
2  c = b'Podstawy programowania w Pythonie' # <class 'bytes'>
3
4  d = c.decode('ASCII')
5
6  if isinstance(d,str):
7    print(type(d))
8  else:
9    print(type(a))
10
11 if (d==a):
12   print ("Kodowanie zakończyło się powodzeniem")
13 else:
14   print ("Kodowanie nie powiodło się")
```

Obiekty klasy `bytearray` są mutowalnymi odpowiednikami obiektów klasy `bytes`. Funkcja `bytearray([source[,encoding[,errors]]])` pozwala utworzyć obiekt klasy `bytearray` w podobny sposób jak funkcja `bytes`:

```
1  bytearray()
2  bytearray(int)
3  bytearray(iterable_of_ints)
4  bytearray(string, encoding[, errors])
5  bytearray(bytes_or_buffer)
```

Na listingach 6.18 i 6.19 przedstawiamy pewne operacje, które można wykonywać zarówno na obiektach typu `bytes`, jak i `bytearray`, czyli: odwoływanie się do elementów obydwu obiektów wypisując je na ekranie (linie 2-3), wypisując fragmenty tych obiektów określone zakresami (linie 5-7). Różnica polega na tym, że przy próbie zmiany elementu obiektu `bytes` (6.18: linia 9) zgłoszony zostanie błąd typu. Gdy tymczasem na listingu 6.19 w liniach 8-14 dokonujemy zamiany wielkości liter tworzących sekwencję `bytearray` uzyskując w efekcie na ekranie napis `bytearray(b'pYTHON')`.

Listing 6.18: *Operacje na obiekcie* `bytes`

```
1  r = b"Python" #<class 'bytes'>
2  for i in r:
3    print(chr(i), end=' ') # P y t h o n
4  print()
```

```
print(r[0:2]) # b'Py'
print(r[0]) # 80
print(r[:-2]) # b'Pyth'

#r[0] = 's'
'''
Traceback (most recent call last):
  File "w10-p7.py", line 8, in <module>
    r[0] = 's'
TypeError: 'bytes' object does not support item assignment
'''
```

Listing 6.19: *Operacje na obiekcie* `bytearray`

```
t = bytearray(b"Python") # <class 'bytearray'>
for i in t:
  print(chr(i), end=' ' ) # P y t h o n
print()
print(t[0:2]) # bytearray(b'Py')
print(t[0]) # 80
print(t[:-2]) # bytearray(b'Pyth')
i = 0
while i < len(t):
  if(chr(t[i]).isupper()):
    t[i] = ord(chr(t[i]).lower())
  else:
    t[i] = ord(chr(t[i]).upper())
  i += 1
print(t) # bytearray(b'pYTHON')
```

W Pythonie 3.x typ `bytes` zawiera sekwencje wartości 8-bitowych, natomiast typ `str` zawiera sekwencje znaków `Unicode`. Z tego też względu przydatne mogą okazać się dwie proponowane przez nas metody:

- metoda `to_str`, która pobiera dane typu `str` lub `bytes` i zawsze zwraca dane w postaci typu `str`;

```
def to_str(bytes_or_str):
    """ to_str """
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode("utf-8")
    else:
        value = bytes_or_str
    return value # zawsze str.
```

- metoda `to_bytes`, która pobiera dane typu `str` lub `bytes` i zawsze zwraca dane w postaci typu `bytes`.

```
1 def to_bytes(bytes_or_str):
2     """ to_bytes """
3     if isinstance(bytes_or_str, str):
4         value = bytes_or_str.encode("utf-8")
5     else:
6         value = bytes_or_str
7     return value # zawsze bytes.
```

6.4.1. Wybrane metody obiektów plików binarnych

Dla plików binarnych dostępne są następujące wybrane metody służące do ich obsługi:

- `f.readlines()` – dla strumienia otwartego w trybie binarnym czyta wszystkie linie z pliku i zwraca listę obiektów klasy `bytes`.
- `f.readline()` – dla strumienia otwartego w trybie binarnym czyta jedną linię i zwraca obiekt klasy `bytes`.
- `f.read(size=-1)` – dla strumienia otwartego w trybie binarnym czyta co najwyżej `size` bajtów ze strumienia (dla `size > 0`) i zwraca obiekt klasy `bytes`. Dla `size < 0` czyta wszystkie bajty od bieżącej pozycji do końca pliku.
- `f.write(buffer)` – dla strumienia otwartego w trybie binarnym zapisuje zawartość bufora `buffer` do pliku. Zwraca liczbę zapisanych bajtów, która jest zawsze długością bufora w bajtach.
- `f.tell()` – zwraca bieżącą pozycję strumienia.
- `f.seek(offset, whence=SEEK_SET)` – ustawia pozycję strumienia. Możliwe wartości drugiego argumentu to:
  - `SEEK_SET` lub 0 – ustawia pozycję względem początku strumienia;
  - `SEEK_CUR` lub 1 – ustawia pozycję względem bieżącej pozycji;
  - `SEEK_END` lub 2 – ustawia pozycję względem końca strumienia.

  Wszystkie te wartości zdefiniowane są w module `io`. Zatem po instrukcji importu w postaci `import io`, należy się do nich odwoływać poprzez w pełni kwalifikowaną nazwę, np. `io.SEEK_SET`.

Dla zilustrowania operacji na plikach binarnych na listingu 6.20 przedstawiamy program służący do kopiowania pliku graficznego w trybie binarnym. Chcąc przetestować działanie programu należy wcześniej przygotować i umieścić w katalogu, w którym znajduje się plik pythonowy, plik graficzny - w naszym przypadku był to plik o nazwie `obraz.png`. Otwieramy ten plik do odczytu w trybie binarnym (linia 2), a następnie otwieramy drugi plik, który ma być jego kopią, tym razem do zapisu w trybie binarnym (linia 3). Odczytujemy cyklicznie po `1kB` danych z pliku wejściowego (linia 6), dopóki nie

zostanie odczytany koniec pliku - spowoduje to wyjście z pętli (linie 7-8) i każdą z nich zapisujemy do pliku wyjściowego (linia 9). Na zakończenie zamykamy obydwa pliki (linie 10 i 11).

Listing 6.20: *Kopiowanie pliku w trybie binarnym*

```
1  def main():
2    f = open("obraz.png", "rb")
3    g = open("kopia.png", "wb")
4    while True:
5      buf = f.read(1024)
6      if len(buf) == 0:
7        break
8      result = g.write(buf)
9    f.close()
10   g.close()
11 if __name__ == "__main__":
12   main()
```

Na listingu 6.21 przedstawiamy program, który korzysta z argumentów wywołania programu, w których znajduje się podana przez wywołującego nazwa pliku, którego rozmiar w bajtach ma zostać obliczony. W sytuacji podania przez użytkownika złej liczby argumentów, na ekran wysyłana jest informacja o prawidłowej formie wywołania programu, a sam program kończy działanie używając w tym celu funkcji `exit` z modułu `sys`. Jeśli plik zostanie zidentyfikowany na dysku (linia 7), wywoływana jest funkcja `f_size`, która otwiera plik o nazwie przekazanej jako argument do odczytu w trybie binarnym i przesuwa wskaźnik pliku na jego koniec (linia 15), odczytywana jest bieżąca pozycja strumienia (linia 16), która jest jednocześnie zwracana, po zamknięciu pliku (linia 17), jako jego rozmiar (linia 18).

Listing 6.21: *Użycie metod* `tell()` *oraz* `seek()`

```
1  import io, sys, os
2
3  def main():
4    if len(sys.argv) != 2:
5      print("Użycie programu: python", sys.argv[0], "plik")
6      sys.exit(1)
7    if os.path.isfile(sys.argv[1]):
8      print("Rozmiar pliku", sys.argv[1] + ":", end='')
9      print(f_size(sys.argv[1]))
10   else:
11     print("Nie znaleziono pliku o nazwie", sys.argv[1])
```

```
12 def f_size(filepath):
13   f = open(filepath, "rb")
14   f.seek(0, io.SEEK_END)
15   size = f.tell()
16   f.close()
17   return size
18
19 if __name__ == "__main__":
20   main()
```

6.5. Zadania do samodzielnego rozwiązania

1. Napisz program, który realizuje następujące zadania:
   - Otwiera plik o nazwie `hello.txt`.
   - Zapisuje w tym pliku komunikat `"Hello, World!"`.
   - Zamyka ten plik.
   - Otwiera ten sam plik ponownie.
   - Wczytuje z pliku komunikat do zmiennej łańcuchowej i wypisuje go.
   - Zamyka ten plik.
2. Napisz program, który:
   - otwiera plik o nazwie podanej przez użytkownika, następnie zapisuje podaną przez użytkownika informację, po czym zamyka plik;
   - otwiera powyższy plik, odczytuje i wypisuje jego zawartość na ekranie, po czym zamyka go.
3. Napisz program, który wczytuje plik tekstowy linia po linii i zapisuje przeczytane linie w pliku wyjściowym poprzedzając je numerem linii. O nazwy plików należy zapytać użytkownika programu na początku programu.
4. Dla każdego z poniższych punktów napisz funkcję, która pobiera jako argumenty łańcuch znaków będący nazwą pliku, próbuje otworzyć ten plik do odczytu, a następnie, używając metody `readline`, czyta z tego pliku kolejne linie. Po przeczytaniu wszystkich linii funkcja zamyka plik. Jako swój wynik funkcja zwraca:
   (a) długość najdłuższej linii z tego pliku;
   (b) najdłuższą linię z tego pliku (jeżeli jest więcej linii o tej samej długości, to funkcja zwraca pierwszą z tych linii);
   (c) krotkę, której pierwszym elementem jest długość najdłuższej linii z tego pliku, a drugim ta linia.

   W przypadku, gdy pliku nie udało się otworzyć, każda z funkcji powinna zwrócić wartość `None`. Przetestuj działanie powyższych funkcji w funkcji `main`.

5. Napisz program `catfiles.py`, który łączy zawartość kilku plików w jeden plik. Przykładowo, `python3 catfiles.py r1.txt r2.txt r3.txt ksiazka.txt` tworzy plik wynikowy `ksiazka.txt`, który zawiera zawartość plików w kolejności `r1.txt`, `r2.txt` i `r3.txt`. Plik docelowy jest zawsze ostatnim plikiem podanym w wierszu poleceń.
6. Napisz program `reverse.py`, który wczytuje wszystkie linie z pliku i zapisuje je w odwrotnej kolejności w pliku wynikowym. Przykładowo, jeżeli plik `input.txt` składa się z linii:

   ```
   Mary had a little lamb
   Its fleece was white as snow
   And everywhere that Mary went
   The lamb was sure to go.
   ```

   to po wywołaniu `python3 reverse.py input.txt output.txt` plik wyjściowy `output.txt` będzie zawierał linie:

   ```
   The lamb was sure to go.
   And everywhere that Mary went
   Its fleece was white as
   Mary had a little lamb
   ```

7. Napisz funkcję `reversed_line`, której argumentem jest łańcuch znaków i która jako wynik zwraca odwrócony łańcuch znaków, przy czym, jeżeli ostatnim znakiem łańcucha jest znak nowej linii (`'\n'`), to pozostawia go na swoim miejscu. Przetestuj tę funkcję w funkcji `main`.
8. Napisz program `reverse_lines.py`, który zamienia każdą linię w pliku jej rewersem. Przykładowo, jeżeli uruchomimy program poprzez
   `python reverse_lines.py hello.py`
   to zawartość pliku `hello.py`

   ```
   1 # My first Python program.
   2 print("Hello, World!")
   ```

   zmieni się na

   ```
   1 .margorp nohtyP tsrif yM #
   2 )"!dlroW ,olleH"(tnirp
   ```

   Po ponownym uruchomieniu programu `reverse_lines.py` na tym samym pliku, odzyskamy oryginalny plik. W programie wykorzystaj funkcję z poprzedniego zadania o nazwie `reversed_line`.

9. Wiadomo, że pierwszych 8 bajtów pliku w formacie `PNG` ma następujące wartości:

   ```
   137, 80, 78, 71, 13, 10, 26, 10
   ```

   Napisz program `is_png.py`, który sprawdza czy plik podany jako jego argument jest obrazem w formacie `PNG`.
10. Uogólnij program z poprzedniego zadania, tak aby można mu było podać w linii wywołania programu więcej argumentów będących nazwami plików.
11. Napisz dwuargumentową funkcję `szyfruj`, której pierwszy argument `buf` jest obiektem typu `bytes`, a drugi argument `mask` jest liczbą całkowitą z przedziału `range(256)`. Funkcja ta zwraca obiekt klasy `bytes`, którego każdy element ma wartość będącą wynikiem zastosowania operatora alternatywy rozłącznej do odpowiadającego elementu z `buf` oraz liczby `mask`. W funkcji `main` przetestuj poprawność działania funkcji `szyfruj`. Na przykład wywołanie `szyfruj(b'\x00\01\x02', 255)` zwróci obiekt `b'\xff\xfe\xfd'`. Z kolei wywołanie `szyfruj(b'Python', 255)` spowoduje zwrócenie obiektu `b'\xaf\x86\x8b\x97\x90\x91'`.
12. Napisz program `szyfr_xor.py` składający się z funkcji `main` oraz z funkcji `szyfruj` z poprzedniego zadania. Funkcja `main` sprawdza czy program został wywołany z trzema argumentami. Jeżeli nie, to wypisuje odpowiedni komunikat i kończy działanie programu. Następnie próbuje otworzyć w trybie binarnym do odczytu plik o nazwie podanej jako pierwszy argument programu, a następnie próbuje otworzyć w trybie binarnym do zapisu plik o nazwie podanej jako drugi argument programu. W przypadku niepowodzenia wypisuje odpowiedni komunikat i kończy wykonanie programu. Ponadto, jeżeli trzeci argument nie jest liczbą z przedziału `range(256)`, to wypisuje odpowiedni komunikat i kończy działanie programu. W przeciwnym przypadku funkcja `main` w pętli czyta z pliku wejściowego kolejne porcje danych o wielkości `32 bajtów`, wywołuje funkcję `szyfruj` dla kolejnej porcji oraz liczby podanej jako trzeci argument programu, po czym zapisuje wynik zwrócony przez funkcję `szyfruj` do pliku wyjściowego.
13. Napisz dwuargumentową funkcję `szyfr_cezara`, której pierwszy argument `buf` jest obiektem typu `bytes`, a drugi argument `shift` jest liczbą całkowitą. Funkcja ta zwraca obiekt klasy `bytes`, którego każdy element ma wartość będącą wynikiem reszty z dzielenia przez 256 sumy odpowiedniego elementu z `buf` oraz liczby `shift`. W funkcji `main` przetestuj poprawność działania funkcji `szyfr_cezara`. Na przykład, wynikiem wywołania `szyfr_cezara(b'\x00\01\x02',3)` powinien być obiekt `b'\x03\x04\x05'`, z kolei wynikiem wywołania `szyfr_cezara(b'Python',-6)` powinien być obiekt `b'Jsnbih'`.
14. Napisz program `szyfr_cezara.py` analogiczny do programu `szyfr_xor.py`, ale z wykorzystaniem funkcji `szyfr_cezara`.

15. Napisz program, który w pętli prosi o podanie liczby naturalnej i zapisuje każdą podaną liczbę do odpowiedniego pliku: parzyste do `even.txt`, a nieparzyste do `odd.txt`. Program kończy się, gdy wpisane zostanie 0. Pliki powinny być zapisywane w nowym folderze, który zostanie utworzony przy uruchomieniu programu (o ile jeszcze nie istnieje). Aby to zrobić, trzeba użyć modułu `os`. Przydatne funkcje:
    - `os.path.exists(mypath)` - sprawdza, czy folder istnieje;
    - `os.mkdir(mypath)` - tworzy folder w zadanej ścieżce `mypath`;
    - `os.getcwd()` - aktualny katalog.

    Uruchomienie programu po raz kolejny powinno powodować, że nowe liczby zostaną dopisane do istniejących plików lub zostaną utworzone nowe pliki.
16. Napisz program, który wczyta pliki utworzone w poprzednim zadaniu i obliczy sumę oraz średnią arytmetyczną liczb znajdujących się w każdym z nich.
17. Na dysku znajdują się trzy katalogi, a w każdym z nich po trzy pliki tekstowe. Napisz program, który wylosuje katalog, a następnie wylosuje jeden ze znajdujących się wewnątrz plików i wypisze jego zawartość na ekranie. Zakładamy, że nazwy katalogów i plików są znane użytkownikowi.
18. Napisz program, który dla podanego przez użytkownika pliku tekstowego wyświetli statystykę składającą się z informacji o liczbie linii w nim zawartych i wszystkich wyrazów występujących w tym pliku.
19. Napisz program, który dla pliku `dane.txt` utworzy i zapisze pod nazwą `dane.cez` zaszyfrowaną wersję tego pliku (można wykorzystać np. szyfr Cezara).
20. Napisz program, który otworzy plik `in.txt`, następnie sprawdzi czy występują w nim ciągi znaków `" ,"` (spacja, przecinek), zamieni je na `", "` (przecinek, spacja) i wynik zapisze w `pliku out.txt`.

Rozdział 7

Programowanie obiektowe

Od samego początku pracy z Pythonem mówimy, że pracujemy na obiektach, które służą do przechowywania i przetwarzania danych w programach komputerowych. Każdy obiekt ma swoją `tożsamość`, `typ` i `stan`. Do tej pory posługiwaliśmy się typami wbudowanymi (czyli dostarczanymi przez standardowo dostępne biblioteki Pythona), teraz, dzięki programowaniu zorientowanemu obiektowo (ang. *Object-Oriented Programming*), będziemy mogli odzwierciedlić elementy świata rzeczywistego (i nie tylko) poprzez definiowanie własnych obiektów.

W tym rozdziale czytelnik zostanie zapoznany z podstawowymi pojęciami związanymi z programowaniem zorientowanym obiektowo, dzięki czemu nie tylko będzie mógł definiować własne typy danych, ale również, stosując modułowość, ściśle związaną z tym paradygmatem, zadbać o lepszą organizację programu.

7.1. Podstawowe pojęcia

Podstawowym pojęciem w programowaniu obiektowym jest `klasa`, która jest szablonem opisującym, jak dany obiekt będzie skonstruowany i jak będzie się zachowywał. Obiekt utworzony na podstawie danej klasy nazywany jest jej `instancją`, a proces jego tworzenia `instancjonowaniem`. W klasie można definiować `atrybuty` obiektu, którymi są zmienne (klasowe i instancyjne) oraz metody (statyczne, klasowe i instancyjne). `Zmienna klasowa` to zmienna, która jest wspólna dla wszystkich instancji klasy i jest definiowana w klasie, ale poza jej metodami. Z kolei `zmienna instancyjna` (inaczej `pole`) to zmienna, która należy tylko do bieżącej instancji klasy i jest definiowana wewnątrz metod.

Do podstawowych własności programowania obiektowego należą:

– `Dziedziczenie` - pozwala na definiowanie nowych klas w oparciu o istniejące, rozszerzając lub zmieniając ich funkcjonalność. Dziedziczenie w Pythonie oparte jest na wyszukiwaniu atrybutów (w wyrażeniach `X.nazwa`).

– `Polimorfizm` - w wyrażeniu `X.metoda` znaczenie atrybutu `metoda` uzależnione jest od typu (klasy) obiektu `X`, co oznacza że zachowanie się metody nie zależy tylko od jej nazwy, ale też od typu (klasy) obiektu.

– `Hermetyzacja` (lub *enkapsulacja*) - metody oraz operatory implementują zachowanie. Ukrywanie danych jest domyślną konwencją.

7.1.1. Definiowanie klas

Instrukcja `class` tworzy obiekt klasy i przypisuje mu nazwę. Jest instrukcją wykonywalną w Pythonie (podobnie jak `def`), co oznacza że kiedy interpreter Pythona napotyka w kodzie instrukcję `class`, wykonuje ją, generuje nowy obiekt klasy i przypisuje go do nazwy podanej w nagłówku instrukcji. Instrukcja `class` zwyczajowo wykonywana jest przy pierwszym importowaniu zawierającego ją pliku. Przypisania wewnątrz instrukcji `class` tworzą atrybuty klasy. Z kolei przypisania na najwyższym poziomie wewnątrz instrukcji `class` (nieosadzone wewnątrz instrukcji `def`) generują atrybuty w obiekcie klasy. Zatem zakres instrukcji `class` staje się przestrzenią nazw atrybutów obiektu, a dostęp do nich jest możliwy za pomocą składni kwalifikowanej `obiekt.atrybut`. Atrybuty klasy udostępniają stan obiektu i jego zachowanie. Ponadto rejestrują informacje o stanie oraz zachowanie współdzielone przez wszystkie instancje utworzone za pomocą tej klasy. Instrukcje `def` zagnieżdżone wewnątrz instrukcji `class` generują metody, które przetwarzają instancje.

Jako przykład zdefiniujemy klasę reprezentującą punkt w układzie współrzędnych, którą przedstawiamy na listingu 7.1.

Listing 7.1: *Definicja klasy* `Punkt`

```
# punkt.py
class Punkt:
  def __init__(self, x, y):
    self._x, self._y = x, y
  def move(self, deltaX, deltaY):
    self._x += deltaX;
    self._y += deltaY
  def __str__(self):
    return f"({self._x}, {self._y})"
```

Pierwszy argument każdej metody, zdefiniowanej wewnątrz klasy `Punkt`, o nazwie `self` jest referencją do obiektu, na rzecz którego ta metoda będzie wywołana. Metody, których nazwy zaczynają się i kończą dwoma znakami podkreślenia `__` to metody **specjalne**. Przykładowo, metoda `__init__` jest automatycznie wywoływana przy tworzeniu obiektu klasy. Ta metoda nie jest jednakże konstruktorem klasy, lecz jej **inicjalizatorem**. Argumenty tej metody, za wyjątkiem pierwszego, służą do inicjalizowania zmiennych instancyjnych: `self._x` i `self._y`. Kolejną metodą specjalną w ciele klasy `Punkt` jest metoda `__str__`, która tworzy reprezentację obiektu w postaci łańcucha znaków. W rezultacie wyświetlenie obiektu pokazuje cokolwiek, co zwróci jego metoda `__str__`. Aby zamienić obiekt `ob` na łańcuch znaków, można wywołać na rzecz obiektu `a` metodę `__str__`: `a.__str__()`. Jednak bardziej naturalnym sposobem jest użycie funkcji wbudowanej `str` z obiektem jako jej argumentem: `str(ob)`. Funkcja `str` zadziała zgodnie z oczekiwaniem, ponieważ wywoła ona metodę `__str__`. Metoda `__str__` wykonywana jest automatycznie za każdym razem, gdy instancja przekształcana jest na swój łańcuch wyświetlania, co oznacza, że jeśli zdefiniujemy metodę `__str__`, to możemy wyświetlać opis obiektu z wykorzystaniem metody `print`, w przeciwnym wypadku opis jest domyślny i zawiera m.in. nazwę klasy oraz adres sektora pamięci, w którym przechowywany jest obiekt.

LISTING 7.2: *Program używający obiektu klasy* `Punkt`

```
# punkt_main_1.py
from punkt import Punkt

def main():
  a = Punkt(5, 8)
  print("Utworzony punkt:", a)
  a.move(-2, 3)
  print("Punkt po przesunięciu: ", a)
  print("Typ utworzonego obiektu:", type(a))
  print("Słownik __dict__ dla utworzonego obiektu:")
  print(a.__dict__)
  b = Punkt(3, 2)
  print("Punkt b:", b)

if __name__ == "__main__":
  main()
```

Czym różni się metoda `__repr__` od `__str__`? Obydwie zwracają reprezentację łańcuchową obiektu klasy, przy czym `__repr__` informuje również o nazwie klasy, której obiekt jest instancją. Wywołujemy ją na rzecz obiektu, np. w następujący sposób `print(repr(a))`.

```
1 def __repr__(self):
2   return  (f'{self.__module__}.'
3            f'{self.__class__.__name__}'
4            f'({self._x}, {self._y})')
```

Wraz z każdym wywołaniem obiektu klasy tworzony i zwracany jest nowy obiekt instancji. Każdy obiekt instancji dziedziczy atrybuty klasy oraz otrzymuje własną przestrzeń nazw. Przypisania do atrybutów `self` w metodach tworzą atrybuty dla poszczególnych instancji. Przypisania do atrybutów `self` tworzą lub modyfikują dane w instancji, a nie w klasie.

Instancje klasy powstają w wyniku wywołania funkcji, której nazwą jest nazwa klasy: `a = Punkt(5, 8)` (listing 7.2: linia 5). Po utworzeniu nowej instancji klasy jej atrybuty i metody dostępne są za pomocą operatora wyboru składowej, czyli kropki (`.`), np. `a.move(-2, 3)` (listing 7.2: linia 7). Wewnętrznie każda instancja jest implementowana przy użyciu słownika `__dict__` zawierającego unikatowe informacje o tej instancji, które można wyświetlić na ekran w następujący sposób: `print(a.__dict__)` (listing 7.2: linia 11).

W klasie możemy zdefiniować również zmienną odnoszącą się do klasy, a nie obiektu, która będzie współdzielona przez wszystkie obiekty tej klasy. Aby utworzyć zmienną klasy, wystarczy ją zdefiniować w ciele klasy. By odnieść się do niej, używamy najpierw nazwy klasy, następnie kropki i wreszcie nazwy tej zmiennej. Nie ma więc potrzeby tworzenia żadnych obiektów, choć gdy już powstaną to na ich rzecz taka zmienna może być wywoływana. Należy jednak pamiętać, że w chwili powstania zmiennej instancji o tej samej nazwie, w przypadku odnoszenia się do zmiennej klasowej poprzez obiekt, Python zawsze będzie identyfikował nazwę zmiennej instancyjnej, a nie klasowej. Prowadzić to może do błędów. Zmienną klasową często wykorzystuje się na przykład do śledzenia liczby utworzonych obiektów danej klasy, stąd w momencie jej definiowania przypisywana jej jest wartość 0, a metody specjalne inicjalizatora i destruktora klasy odpowiednio ją zwiększają i zmniejszają o 1. W dalszej części rozdziału zaprezentujemy praktyczne wykorzystanie zmiennej klasowej - podrozdział 7.4.

7.1.2. Hermetyzowanie nazw w klasie

Jedną z podstawowych własności programowania obiektowego jest hermetyzowanie danych w obiektach danej klasy, przy czym sam język Python nie dostarcza mechanizmów temu służących. Istnieją obowiązujące konwencje nazewnicze związane z przeznaczeniem danych i metod. Konwencje te stanowi, iż każda nazwa rozpoczynająca się od znaku jednego podkreślenia (`_`) oznacza jednostkę wewnętrzną. Python nie blokuje dostępu do jednostek wewnętrznych. Jednak korzystanie z nich jest uznawane za nieeleganckie i może

prowadzić do powstania kodu podatnego na błędy, na przykład poprzez nieopatrzne usunięcie atrybutów na zewnątrz klasy zilustrowanie na listingu 7.3 w linii 7.

Listing 7.3: *Usunięcie atrybutu klasy* `Punkt`

```
1  # punkt_main_2.py
2  from punkt import Punkt
3
4  def main():
5    a = Punkt(5, 8)
6    print("Utworzony punkt:", a)
7    del a._y
8    try:
9      print("Po usunięciu atrybutu _y:", a)
10   except AttributeError as ex:
11     print("\n", ex.args)
12
13 if __name__ == "__main__":
14   main()
```

O fakcie usunięcia atrybutu klasy `Punkt`, do którego następnie próbujemy się odwołać w zdefiniowanej metodzie specjalnej `__str__` zostaniemy poinformowani w wyświetlanym na ekranie komunikacie, a dzięki użyciu obsługi wyjątków program zakończy się bez przerwania z powodu jego wystąpienia:

```
Utworzony punkt: (5, 8)
Po usunięciu atrybutu _y:
 ("'Punkt' object has no attribute '_y'",)
```

Większość języków, aby zabezpieczyć dostęp do składowych klasy, definiuje te składowe jako prywatne za pomocą słowa kluczowego `private` lub podobnego. Prywatne zmienne i metody są przydatne z dwóch powodów:

1. zwiększają bezpieczeństwo i stabilność kodu dzięki selektywnemu odmawianiu dostępu do ważnych czy delikatnych części implementacji obiektu; jasno określają co jest używane wewnętrznie przez klasę;
2. pozwalają uniknąć konfliktów nazw wynikających z dziedziczenia; dzięki temu, że są one prywatne, każda klasa ma własne ich kopie.

Nazwy zmiennych instancji i metod prywatnych w Pythonie, zgodnie z obowiązującą konwencją, zaczynają się od podwójnego znaku podkreślenia, ale się nimi nie kończą. Ani zmienna prywatna, ani metoda prywatna nie są widzialne poza metodami klasy, w której zostały zdefiniowane. Zastosowany mechanizm zapewniania prywatności zniekształca nazwy prywatnych zmiennych i metod, gdy kod zostanie skompilowany do kodu pośredniego

(ang. *bytecode*) poprzez dodanie na początku do tych nazw nazwy klasy ze znakiem podkreślenia, np. stosując funkcję `dir`, która zwraca listę atrybutów i metod danego obiektu, możemy zobaczyć atrybuty prywatne instancji klasy `Punkt` (specjalnie nie pokazujemy innych jej atrybutów i metod, skupiając się na tych dla nas istotnych):

```
>>> dir(Punkt(3, 5))
['_Punkt__x', '_Punkt__y', ...]
```

Operacja ta nazywa się **dekorowaniem nazw** (ang. *name mangling*, *name decoration*), a jej celem jest zapobiegnięcie jakimkolwiek przypadkowym udostępnieniom zmiennej. Można oczywiście celowo „zasymulować" dekorowanie jakie nastąpi i w ten sposób uzyskać dostęp do zmiennej, np. `a._Punkt__x = 111`. Dokonanie dekorowania we wskazanej wyżej formie ułatwia debugowanie programu.

Na listingu 7.4 przedstawiamy wersję definicji klasy `Punkt`, w której zmienne instancji zostały oznaczone jako prywatne. Dzięki temu modyfikowanie pól na zewnątrz klasy nie jest już możliwe.

LISTING 7.4: *Klasa* `Punkt` *z polami prywatnymi*

```
# punkt_priv.py
class Punkt:
  def __init__(self, x, y):
    self.__x, self.__y = x, y

  def move(self, deltaX, deltaY):
    self.__x += deltaX; self.__y += deltaY

  def __str__(self):
    return f"({self.__x}, {self.__y})"

# main_priv.py
from punkt_priv import Punkt

def main():
  a = Punkt(5, 8)
  print("Utworzony punkt:", a)
  print(a.__dict__)
  a.__x = 3
  print("Po instrukcji a.__x = 3:", a)
  print(a.__dict__)

if __name__ == "__main__":
  main()
```

```
Utworzony punkt: (5, 8)
{'_Punkt__x': 5, '_Punkt__y': 8}
Po instrukcji a.__x = 3: (5, 8)
{'_Punkt__x': 5, '_Punkt__y': 8, '__x': 3}
```

W sytuacji, gdy odwołamy się do zmiennej prywatnej instancji (linia 19), ale bez użycia dekoratora nazw, spowoduje to tylko powiększenie słownika `__dict__`, będącego przestrzenią nazw obiektu zawierającą jego atrybuty, o nowy atrybut `'__x'`.

Jak już wspomnieliśmy wyżej, z każdą instancją klasy związany jest słownik `__dict__` przechowujący jej atrybuty. Powoduje to niegospodarne wykorzystanie pamięci szczególnie w przypadku obiektów mających niewielką liczbę zmiennych poziomu instancji, ale których powstaje bardzo duża liczba w programie.

W klasach, które przede wszystkim pełnią funkcję prostych struktur danych, często można znacznie zmniejszyć ilość pamięci zajmowaną przez obiekty dodając do definicji klasy atrybut `__slots__`. Jest on zmienną poziomu klasy, której można przypisać nazwy zmiennych używanych przez instancje, które mogą być napisem, obiektem iterowalnym lub sekwencją napisów. W poniższej definicji klasy `Date`, atrybut `__slots__` jest krotką nazw jej prywatnych atrybutów przechowujących informacje o roku, miesiącu i dniu.

```
1 # date.py
2 class Date:
3   __slots__ = ('__year', '__month', '__day')
4   def __init__(self, year, month, day):
5     self.__year = year
6     self.__month = month
7     self.__day = day
```

Gdy zdefiniujemy atrybut `__slots__`, Python będzie stosował znacznie zwięźlejszą reprezentację obiektów. Zamiast dodawać słownik do każdego obiektu, Python tworzy wtedy obiekty oparte na małej tablicy o stałym rozmiarze przypominającej krotkę lub listę. Nazwy atrybutów wymienione w specyfikatorze `__slots__` są wewnętrznie odwzorowywane na konkretne indeksy tablicy. Efektem ubocznym stosowania tej techniki jest to, że do obiektów nie można dodawać nowych atrybutów - dozwolone są tylko te wymienione w specyfikatorze `__slots__`. Choć może się wydawać, że przedstawione rozwiązanie jest przydatne w wielu sytuacjach, nie należy go nadużywać. W wielu miejscach Pythona stosuje się standardowy kod oparty na słownikach. Ponadto klasy utworzone za pomocą opisanej techniki nie obsługują niektórych mechanizmów, np. `wielodziedziczenia`. Dlatego technikę tę należy stosować tylko w tych klasach, które są często używane w programie, np. gdy program tworzy miliony obiektów danej klasy. Często technika slotów jest traktowana jako narzędzie zapewniające hermetyzację, które uniemożliwia użytkownikom dodawanie nowych atrybutów do obiektów.

W przypadku instancji, które nie posiadają atrybutu `__dict__`, nie jest możliwe przypisywanie zmiennych, które nie zostały wymienione w definicji `__slots__`. Próba wykonania takiego przypisania spowoduje wygenerowanie wyjątku `AttributeError`.

```
>>> d._a = 1
Traceback (most recent call last):
  File "mainDate.py", line 22, in <module> main()
  File "mainDate.py", line 9, in main d._a = 1
AttributeError: 'Date' object has no attribute '_a'
```

Jeśli w programie wymagana jest możliwość dynamicznego dodawania nowych zmiennych, należy dodać nazwę `'__dict__'` do sekwencji napisów przypisanej atrybutowi `__slots__` (`__slots__ = ('__year', '__month', '__day', '__dict__')`). Przy czym zwracamy uwagę, że w słowniku tym nie zobaczymy atrybutów wymienionych w `__slots__`.

```
>>> d._a = 1
>>> d.__dict__
{'_a': 1}
```

Pamiętajmy, że działanie atrybutu `__slots__` ograniczone jest do klasy, w której został on zdefiniowany. Dlatego klasy pochodne będą w zwykły sposób zawierać atrybut `__dict__`, chyba że w nich również zostanie zdefiniowany `__slots__`.

7.2. Dziedziczenie

Dziedziczenie to mechanizm tworzenia nowych klas rozszerzających już istniejące. Klasa, po której dziedziczymy, nazywa się klasą **`bazową`** lub **`nadklasą`** (czasem też „rodzicem"), zaś klasa dziedzicząca nazywana jest klasą **`pochodną`** lub **`podklasą`** (lub „dzieckiem"). Podklasa dziedziczy wszystkie atrybuty oraz metody nadklasy, szczególnym przypadkiem odziedziczonej metody jest inicjalizator `__init__`. Inicjalizator klasy pochodnej przyjmuje jako argumenty: jako pierwszy występuje `self`, następnie argumenty niezbędne do inicjalizacji atrybutów klasy bazowej, a na końcu argumenty inicjalizujące atrybuty w klasie pochodnej. Wymieniona kolejność jest umowna, pozwala jednak na kontrolę poprawności. W ciele inicjalizatora wykonujemy oryginalną metodę `__init__` klasy `Punkt`, wywołując ją za pośrednictwem nazwy klasy i przekazując jej `self` oraz pozostałe argumenty w sposób jawny. Python wykorzystuje dziedziczenie do wyszukania i wywołania tylko jednej metody `__init__` naraz w czasie tworzenia instancji, tej znajdującej się najniżej w drzewie klas. Chcąc wykonać metodę `__init__` znajdującą się wyżej w drzewie dziedziczenia, musimy wywołać ją ręcznie za pośrednictwem nazwy klasy nadrzędnej. Zaletą tego rozwiązania jest pozostawienie programiście do decyzji czy i jak wywoływać inicjalizator klasy nadrzędnej.

Dziedziczenie zapisuje się w instrukcji `class` za nazwą klasy pochodnej przy pomocy listy ujętej w nawiasy okrągłe, w której wymienia się oddzielone przecinkami nazwy klas bazowych. Jako przykład zdefiniujemy rozszerzenie klasy `Punkt` z listingu 7.1 poprzez zdefiniowanie klasy pochodnej `NazwanyPunkt` (listing 7.5). Rozszerzenie to wprowadza dodatkowy atrybut przechowujący nazwę punktu (linia 6). W klasie pochodnej można nadpisać wybrane metody klasy bazowej. Jako przykład zdefiniujemy w klasie pochodnej jej inicjalizator `__init__` (linie 4-6) oraz metodę `__str__` (linie 7 i 8). W linii 5 definicji inicjalizatora wywoływany jest jawnie inicjalizator klasy bazowej.

Listing 7.5: *Definicja klasy pochodnej* `NazwanyPunkt`

```
1 # namepunkt.py
2 from punkt_priv import Punkt
3 class NazwanyPunkt(Punkt):
4   def __init__(self, x, y, nazwa):
5     Punkt.__init__(self, x, y)
6     self.__nazwa = nazwa
7   def __str__(self):
8     return f"{self.__nazwa}" + Punkt.__str__(self)
```

Na listingu 7.6 tworzymy obiekt `a` (linia 5), jako punkt o współrzędnych `(5,8)`, któremu nadano nazwę `"A"`. Wypisujemy na ekran dane o utworzonym punkcie i tu Python wybierze metodę `__str__` redefiniowaną w klasie potomnej (linia 6). Obiekt ten odziedziczył po klasie `Punkt` wszystkie jej atrybuty i metody, dlatego przesuwamy punkt `a` w układzie współrzędnych wykorzystując w tym celu metodę `move` klasy `Punkt` (linia 7). Ponownie wypisujemy na ekran wartości punktu `a` po przesunięciu. W linii 10 tworzymy jeszcze obiekt klasy `Punkt` i wypisujemy informacje o nim na ekran (linia 11).

Listing 7.6: *Definiowanie obiektów klas: bazowej i pochodnej*

```
1 # Namepunkt_main.py
2 from punkt_priv import Punkt
3 from nazwanypunkt import NazwanyPunkt
4 def main():
5   a = NazwanyPunkt(5, 8, "A")
6   print('Utworzono NazwanyPunkt:', a)
7   a.move(-2, 3)
8   print('NazwanyPunkt po przesunięciu:', a)
9   print('NazwanyPunkt.__dict__:\n', a.__dict__)
10  b = Punkt(3, 4)
11  print('Utworzono Punkt:', b)
12 if __name__ == "__main__":
13   main()
```

W wyniku wywołania programu z listingu 7.6 na ekranie zobaczymy informacje:

```
Utworzono NazwanyPunkt: A(5, 8)
NazwanyPunkt po przesunięciu: A(3, 11)
NazwanyPunkt.__dict__:
{'_Punkt__x': 3, '_Punkt__y': 11, '_NazwanyPunkt__nazwa': 'A'}
Utworzono Punkt: (3, 4)
```

Klasa `Punkt` jest podklasą siebie samej:

```
>>> from punkt import Punkt
>>> from nazwanypunkt import NazwanyPunkt
>>> issubclass(Punkt, Punkt)
True
```

Ale nie jest podklasą klasy `NazwanyPunkt`:

```
>>> issubclass(Punkt, NazwanyPunkt)
False
```

Natomiast klasa `NazwanyPunkt` jest podklasą klasy `Punkt`:

```
>>> issubclass(NazwanyPunkt, Punkt)
True
```

Teraz tworzymy dwa obiekty: jeden klasy `Punkt` i drugi klasy `NazwanyPunkt` , dla których sprawdzamy wzajemne zależności.

```
>>> a = NazwanyPunkt(5, 8, "A")
>>> b = Punkt(3, 5)
```

Czy obiekt `a` jest instancją klasy `Punkt`?

```
>>> isinstance(a, Punkt)
True
```

Czy obiekt `a` jest instancją klasy `NazwanyPunkt`?

```
>>> isinstance(a, NazwanyPunkt)
True
```

Czy obiekt `b` jest instancją klasy `NazwanyPunkt`?

```
>>> isinstance(b, NazwanyPunkt)
False
```

Każda klasa w języku Python dziedziczy bezpośrednio lub pośrednio po klasie `object`, a tym samym dziedziczy jej metody. Stąd nagłówek `class NazwaKlasy` definicji klasy jest równoważny nagłówkowi `class NazwaKlasy(object)`. Aby sprawdzić co udostępnia nam klasa `object` możemy uruchomić w interpreterze funkcję `dir` dla obiektu tej klasy zwracającą listę jego atrybutów.

```
>>> ob = object()
>>> dir(ob)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Możliwość dziedziczenia z podklas daje nam możliwość tworzenia bogatych hierarchii dowolnie wyspecjalizowanych klas o strukturze drzewa, w którym krawędzie ustawione są „na odwrót", tzn. strzałki prowadzą od podklas do ich bezpośrednich klas bazowych. W przypadku omawianego w tym rozdziale dziedziczenia będzie to drzewo postaci:

`Punkt` ← `NazwanyPunkt`

Gdy wywołujemy metodę na obiekcie danego typu, metoda ta jest wyszukiwana najpierw w klasie tego obiektu, następnie jego klasie bazowej (jeśli taka jest), później jej klasie bazowej, i tak dalej, odwiedzając klasy coraz wyżej w hierarchii dziedziczenia, aż do znalezienia metody lub wyczerpania klas.

W przykładzie z listingu 7.6 dla obiektu `a`, metoda `move` szukana będzie po kolei w klasach `NazwanyPunkt, Punkt` i zostanie znaleziona w klasie `Punkt`. W przypadku metody `__str__` zostanie ona już znaleziona w klasie `NazwanyPunkt`, stąd była potrzeba zdefiniowania w jej ciele wywołania odpowiedniej jej wersji z klasy bazowej. Mechanizm ten pozwala na tworzenie dowolnie wyspecjalizowanych klas obiektów, realizujących indywidualne operacje na własny sposób, mogący wpływać również na realizacje algorytmów (operacji) zadanych w ich nadklasach. Wszystko przy zachowaniu ścisłej hierarchii typów, pozwalającej na łatwą weryfikację, jakie operacje są dostępne dla konkretnego obiektu.

Identyczny efekt uzyskamy używając wbudowanego obiektu `super`:

```
1 class NazwanyPunkt(Punkt):
2
3   def __init__(self, x, y, nazwa):
4     super().__init__(x, y)
5     self.__nazwa = nazwa
6
7   def __str__(self):
8     return f"{self.__nazwa}"+ super().__str__()
```

Obiekt `super` utworzony w implementacji metody klasy będzie zachowywał się tak jak `self`, ale dla takiego obiektu inna będzie kolejność przeszukiwania metod: będą one szukane począwszy od klasy bazowej danej klasy. W tej sytuacji, `super().__str__()`

stara się wyszukać metody `__str__` począwszy od klasy `Punkt`, nie `NazwanyPunkt` i wywołać ją dla parametru `self`. Klasa może dziedziczyć po więcej niż jednej klasie, co zapisujemy w nagłówku instrukcji `class` wymieniając każdą z klas nadrzędnych w nawiasie i oddzielając je przecinkami. W przypadku takiego typu dziedziczenia, kolejność podawania klas nadrzędnych w nagłówku instrukcji `class` może być kwestią kluczową. Dziedziczenie wielorakie jest już bardzo zaawansowanym zagadnieniem programowania w Pythonie i wykracza poza zakres tego podręcznika. Zainteresowanego czytelnika odsyłamy do dokumentacji Pythona lub pozycji książkowych wymienionych w spisie literatury tego podręcznika.

Wykorzystanie funkcji `super` dotyczy raczej drzewa pojedynczego dziedziczenia i zaczyna rodzić problemy, gdy tylko w tradycyjnie kodowanych klasach zaczyna być stosowane wielokrotne dziedziczenie. W trybie pojedynczego dziedziczenia funkcja może maskować późniejsze problemy i w miarę wzrostu drzewa skutkować nieoczekiwanym działaniem kodu. Funkcja `super` nie zgłosi wyjątku w drzewie wielokrotnego dziedziczenia, tylko w naturalny sposób wybierze pierwszą z lewej klasę nadrzędną posiadającą metodę przeznaczoną do wywołania, która może, ale nie musi być żądaną metodą. W takim przypadku zaleca się jawnie wybierać klasę nadrzędną. Stosowanie funkcji `super` nie jest zabronione, wręcz przeciwnie, stosować ją trzeba, ale z umiarem i w pełni świadomie, po głębokiej analizie i zrozumieniu zaawansowanych zagadnień programowania w Pythonie.

7.3. Metody statyczne i klasowe

Metoda `klasowa` to funkcja zdefiniowana w zakresie klasy, ale poprzedzona dekoratorem `@classmethod`, której pierwszy argument jest zwyczajowo nazywany `cls` i jest referencją do klasy. Metoda `instancyjna` (w skrócie: `metoda`) to funkcja zdefiniowana w zakresie klasy, a jej pierwszy argument jest zwyczajowo nazywany `self` i jest referencją do obiektu, na rzecz którego wywołano tę metodę. Ponadto w klasie istnieją metody, które nie operują na konkretnej instancji klasy, które nazywa się `statycznymi`. W Pythonie nie posiadają one parametru `self`, lecz są opatrzone dekoratorem `@staticmethod`. Statyczną metodę można wywołać zarówno przy pomocy nazwy klasy, jak i jej obiektu i w obu przypadkach rezultat będzie ten sam. Technicznie jest to bowiem zwyczajna funkcja umieszczona po prostu w zasięgu klasy zamiast w zasięgu globalnym.

Na listingu 7.7 przedstawiamy przykład klasy `Date` reprezentującej datę z wyodrębnionymi polami: rok (`y`), miesiąc (`m`) i dzień (`d`). W jej ciele zdefiniowano również dwie metody statyczne. Metoda `today` konwertuje bieżący znacznik czasu lokalnego do formatu klasy `Date`. Z kolei druga metoda `tomorrow` konwertuje bieżący znacznik czasu lokalnego po zwiększeniu o dobę do formatu klasy `Date`.

Listing 7.7: *Przykład klasy* `Date` *z metodami statycznymi*

```
# date.py
import time
class Date(object):
  def __init__(self, y, m, d):
    self._y, self._m, self._d = y, m, d

  @staticmethod
  def today():
    t = time.localtime()
    return Date(t.tm_year, t.tm_mon, t.tm_mday)

  @staticmethod
  def tomorrow():
    t = time.localtime(time.time() + 24 * 60 * 60)
    return Date(t.tm_year, t.tm_mon, t.tm_mday)
```

Następnie definiujemy klasę pochodną `PolishDate` (listing 7.8) dziedziczącą po klasie `Date`, która posiada metodę formatującą datę do postaci `'yyyy-mm-dd'`.

Listing 7.8: *Przykład klasy pochodnej* `PolishDate` *po klasie* `Date`

```
#polishdate.py
from date import Date
class PolishDate(Date):
  def __str__(self):
    s = "{:04}-{:02}-{:02}"
    return s.format(self.y, self.m, self.d)
```

Listing 7.9: *Przykład użycia obiektów klasy* `PolishDate`

```
#mainDate.py
from polishdate import PolishDate
def main():
  d = PolishDate.today()
  print(type(d))
  print("Dzisiaj jest", d)
if __name__ == '__main__':
  main()
```

```
<class 'date.Date'>
Dzisiaj jest <date.Date object at 0x7f0ebebab6a0>
```

Wyniki działania programu z listingu 7.9 nie są tym czego chcielibyśmy i powinniśmy się spodziewać. Jest to spowodowane tym, że umieszczone w klasie `Date` metody

`today` i `tomorrow` są metodami statycznymi. Aby jako wynik na ekranie pojawiła się prawidłowo sformatowana data, powyższe dwie metody powinny być metodami klasowymi. Natomiast przydatną metodą statyczną w klasie `Date` byłaby metoda `seconds_per_day`, która zwraca liczbę sekund doby i może być wykorzystana w metodach klasowych.

```
class Date: ...
  @staticmethod
  def seconds_per_day():
    return 24 * 60 * 60
```

Metody `klasowe` są wywoływane na rzecz całej klasy, a nie jakiejś jej instancji i przyjmują tę klasę jako swój pierwszy parametr. Argument ten jest często nazywany `cls`, ale jest to o wiele słabsza konwencja niż ta dotycząca `self`. W celu odróżnienia od innych rodzajów, metody klasowe oznaczone są dekoratorem `@classmethod`. Podobnie jak metody statyczne, można je wywoływać na dwa sposoby – przy pomocy klasy lub obiektu – ale w obu przypadkach do `cls` trafi wyłącznie klasa. W ogólności może to być także klasa pochodna.

Na listingu 7.10 przedstawiamy poprawioną wersję klasy `Date` z użyciem metod klasowych, dzięki czemu w wyniku wywołania programu z listingu 7.9 na ekranie pojawi się data w prawidłowym i oczekiwanym formacie.

```
<class '__main__.PolishDate'>
Dzisiaj jest 2024-06-25
```

Listing 7.10: *Klasa* `Date` *z metodami klasowymi*

```
# date.py
from time import localtime
class Date(object):
  def __init__(self, y, m, d):
    self._y, self._m, self._d = y, m, d
  @classmethod
  def today(cls):
    t = localtime()
    return cls(t.tm_year, t.tm_mon, t.tm_mday)
  @classmethod
  def tomorrow(cls):
    t = localtime(time.time() + seconds_per_day())
    return cls(t.tm_year, t.tm_mon, t.tm_mday)
  @staticmethod
  def seconds_per_day():
    return 24*60*60;
```

7.4. Przeciążanie operatorów

Klasy pozwalają na `przeciążanie operatorów`, które polega na przechwytywaniu i implementowaniu działania operacji na typach wbudowanych poprzez definiowanie w ciele klasy metod o specjalnych nazwach. Każda z nich rozpoczyna się i kończy dwoma znakami podkreślenia. Nazwy te nie są zarezerwowane i mogą być dziedziczone z klas nadrzędnych w zwykły sposób. Python wyszukuje i wywołuje co najmniej jedną taką metodę w każdej operacji, a w przypadku gdy egzemplarze znajdują się w wyrażeniach oraz innych kontekstach robi to automatycznie. W klasach dopuszczalne jest mieszanie metod przetwarzających liczby i kolekcje oraz działań mutowalnych i niemutowalnych. Większość nazw przeciążających operatory nie ma wartości domyślnych, a odpowiadające im działania zgłaszają wyjątek, jeśli określona metoda nie jest zdefiniowana. Przeciążanie operatorów powinno być wykonywane w klasie tylko wtedy, gdy jest to bezwzględnie wymagane dla zachowania spójności operacji, jak dla typów wbudowanych. W przeciwnym przypadku polecamy stosowanie zwykłych metod. Najczęściej przeciążanie operatorów stosuje się przy definiowaniu struktur matematycznych.

W podrozdziale tym pokażemy na wybranych operatorach wbudowanych jak je przeciążyć na rzecz definiowanej przez użytkownika klasy. Jako przykład zdefiniujemy klasę `Vector`, reprezentującą `n`-wymiarowy wektor będący listą `n` liczb całkowitych reprezentujących jego składowe (tj. `[x_1,x_2,...,x_n]`, gdzie liczby `x_i` to składowe wektora) oraz podstawowe operacje wykonywane na nim.

Rozpoczniemy od zdefiniowania inicjalizatora klasy `Vector`, który tworzy początkowo pustą listę swoich składowych (`_vect`) i kopiuje do niej elementy listy `lista` przekazanej jako trzeci argument. Polu `_rozmiar` przypisywana jest wartość przekazana jako drugi argument, o ile jest ona równa co do wartości długości przekazanej listy. Ponadto w obrębie definicji klasy `Vector` zdefiniowano zmienną klasową `_ile`, która będzie na bieżąco kontrolować liczbę utworzonych instancji klasy.

Listing 7.11: *Definicja klasy* `Vector`

```
1 class Vector:
2   _ile = 0
3   def __init__(self, rozmiar = 0, lista = []):
4     if (n:=len(lista)) != rozmiar: self._rozmiar = n
5     else: self._rozmiar = rozmiar
6     self._vect = []
7     for i in lista:
8       self._vect.append(i)
9     Vector._ile += 1
```

7.4.1. Metody do porównywania obiektów

`Metody bogatych porównań` są wywoływane w odniesieniu do wszystkich wyrażeń wykorzystujących porównania. Należą do nich:

```
__lt__(self, other)  # self < other
__le__(self, other)  # self <= other
__eq__(self, other)  # self == other
__ne__(self, other)  # self != other
__gt__(self, other)  # self > other
__ge__(self, other)  # self >= other
```

Powyższe metody mogą zwrócić dowolną wartość, ale jeżeli operator porównania zostanie użyty w kontekście operacji logicznych, to zwrócona wartość będzie zinterpretowana jako logiczny wynik (typu `Boolean`) działania operatora. Metody te mogą również zwracać (choć nie zgłaszają wyjątku) specjalny obiekt `NotImplemented` w przypadku, gdy operandy ich nie obsługują. Efekt jest taki, jakby metoda w ogóle nie została zdefiniowana. Nie istnieją domniemane relacje pomiędzy operatorami porównań, na przykład z faktu, że wyrażenie `x == y` ma wartość `True`, nie wynika, że wyrażenie `x != y` ma automatycznie wartość `False`. Aby operatory działały symetrycznie, należy zdefiniować metodę `__ne__` razem z metodą `__eq__`. Nie istnieją również prawostronne (o zamienionych argumentach) wersje tych metod do wykorzystania w sytuacji, kiedy lewy argument nie obsługuje określonego działania, a prawy je obsługuje. Metody `__lt__` i `__gt__`, `__le__` i `__ge__` oraz `__eq__` i `__ne__` są swoimi wzajemnymi odbiciami. W Pythonie 3.x dla potrzeb operacji sortowania należy używać metody `__lt__`.

Dwa wektory są różne jeżeli są różne ich wymiary, a także, gdy ich rozmiary są równe, ale różnią się przynajmniej jedną składową.

```
1 def __ne__(self, other):
2   if self._rozmiar != other._rozmiar:
3     return True
4   for i in range(self._rozmiar):
5     if self._vect[i] != other._vect[i]:
6       return True
7   return False
```

7.4.2. Podstawowe metody działań dwuargumentowych

Jeśli któraś z poniższych metod działań dwuargumentowych nie obsługuje działania dla przekazanych argumentów, to powinna zwrócić (nie zgłosić) wbudowany obiekt o nazwie `NotImplemented`, który działa tak, jakby metoda w ogóle nie była zdefiniowana.

Do podstawowych metod działań dwuargumentowych zaliczamy:

```
__add__(self, other)  # self + other
Realizuje dodawanie liczb lub konkatenację sekwencji.
```

```
def __add__(self, other):
  if !isinstace(other,Vector):
    return NotImplemented
  if self._rozmiar != other._rozmiar:
    raise ValueError('Różne rozmiary wektorów!')
  V = Vector(self._rozmiar, self._vect)
  for i in range(self._rozmiar):
    V._vect[i] += other._vect[i]
  return V
```

```
__sub__(self, other)  #  self - other
Realizuje odejmowanie liczb sekwencji.
```

```
__mul__(self, other)  #  self * other
Realizuje mnożenie liczb lub powtarzanie (powielanie) sekwencji.
```

```
#mnożenie przez skalar każdego elementu sekwencji
def __mul__(self, alpha):
  V = Vector(self._rozmiar)
  for i in range( self._rozmiar):
    V._vect.append(self._vect[i])
  for i in range(self._rozmiar):
    V._vect[i] *= alpha
  return V
```

```
__truediv__(self, other)  #  self / other
Realizuje dzielenie rzeczywiste liczb.
```

W celu przeprowadzenia dzielenia (z uwzględnieniem reszty) korzystamy z następującego operatora:

```
__floordiv__(self, other) #  self // other
```

W celu realizacji dzielenia z obcinaniem (część całkowita z dzielenia) korzystamy z jednego z dwóch następujących operatorów:

```
__mod__(self, other)      #  self % other
__divmod__(self, other)   #  divmod(self, other)
```

Dla sekwencji definiowanej w klasie `Vector` możemy również przeciążyć operator potęgowania:

```
__pow__(self, other])     #  pow(self, other) | self ** other
```

Możemy również wykorzystać do przeciążania operatory bitowe takie, jak odpowiednio przesunięcie, koniunkcja, alternatywa wykluczająca czy alternatywa:

```
__lshift__(self, other)   #  self << other
__rshift__(self, other)   #  self >> other
__and__(self, other)      #  self & other
__xor__(self, other)      #  self ^ other
__or__(self, other)       #  self | other
```

7.4.3. Metody prawostronnych działań dwuargumentowych

Nazwy prawostronnych odpowiedników operatorów dwuargumentowych, opisanych w podrozdziale 7.4.2, rozpoczynają się od prefiksu `r`, np. prawostronnym odpowiednikiem operatora `__add__` jest `__radd__`. Odmiany prawostronne mają takie same listy argumentów, ale argument `other` występuje po lewej stronie operatora. Na przykład działanie `self + other` wywołuje metodę `self.__add__(other)`, natomiast `other + self` wywołuje metodę `self.__radd__(other)`.

Metody prawostronne `(r)` są wywoływane tylko wtedy, kiedy egzemplarz klasy znajduje się po prawej stronie, a lewy operand nie jest egzemplarzem klasy, która implementuje działanie:

`egzemplarz + otherobiekt` uruchamia metodę `__add__`

`egzemplarz + egzemplarz` uruchamia metodę `__add__`

`otherobiekt + egzemplarz` uruchamia metodę `__radd__`.

Jeśli w działaniu występują obiekty dwóch klas przeciążających działanie, to preferowana jest klasa argumentu występującego po lewej stronie.

Metoda `__radd__` często jest implementowana w ten sposób, że zamienia kolejność operandów i wywołuje metodę `__add__`. Do metod prawostronnych działań dwuargumentowych należy operator prawostronnego dodawania:

```
__radd__(self, other)      # other + self
```

```
1 def __radd__(self, other):
2   if isinstance(other,int):
3     V = Vector(self._rozmiar)
4     for i in range(self._rozmiar):
5       V._vect.append(self._vect[i])
```

```
6       for i in range(self._rozmiar):
7         V._vect[i] += other
8       return V
9     elif isinstance(other, list):
10      V = Vector(len(other), other)
11      return V + self
12    else:
13      return NotImplemented
```

Zachęcamy czytelnika do samodzielnego zdefiniowania pozostałych wymienionych niżej prawostronnych działań dwuargumentowych

```
__rsub__(self, other)      # other - self
__rmul__(self, other)      # other * self
__rtruediv__(self, other)  # other / self
__rfloordiv__(self, other) # other // self
__rmod__(self, other)      # other % self
__rdivmod__(self, other)   # divmod(self, other)
__rpow__(self, other)      # pow(other, self) | other ** self
__rlshift__(self, other)   # other << self
__rrshift__(self, other)   # other >> self
__rand__(self, other)      # other & self
__rxor__(self, other)      # other ^ self
__ror__(self, other)       # other | self
```

7.4.4. Metody działań dwuargumentowych z aktualizacją w miejscu

Metody działań dwuargumentowych z aktualizacją w miejscu wywoływane są odpowiednio dla następujących formatów instrukcji przypisania:
`+=, -=, *=, /=, //=, %=, **=, <<=, >>=, &=, ^= oraz |=`.
Metody te powinny podejmować próbę wykonania działania w miejscu (z modyfikacją egzemplarza `self`) i zwracać wynik, którym może być egzemplarz `self`. Jeśli metoda jest niezdefiniowana, to działanie z aktualizacją sięga do zwykłych metod. W celu obliczenia wartości wyrażenia `X += Y`, gdzie `X` jest egzemplarzem klasy ze zdefiniowaną metodą `__iadd__`, wywoływana jest metoda `x.__iadd__(y)`. W przeciwnym razie wykorzystywane są metody `__add__` oraz `__radd__`.

Metody przypisania z aktualizacją w miejscu to:

```
__iadd__(self, other)       #  self += other
```

```
def __iadd__(self, other):
  if self._rozmiar == other._rozmiar:
    for i in range(self._rozmiar):
      self._vect[i] += other._vect[i]
    return self
  else:
    raise ValueError('Różne rozmiary wektorów!')
```

```
__isub__(self, other)       #  self -= other
__imul__(self, other)       #  self *= other
__itruediv__(self, other)   #  self /= other
__ifloordiv__(self, other)  #  self //= other
__imod__(self, other)       #  self %= other
__ipow__(self, other)       #  self **= other
__ilshift__(self, other)    #  self <<= other
__irshift__(self, other)    #  self >>= other
__iand__(self, other)       #  self &= other
__ixor__(self, other)       #  self ^= other
__ior__(self, other)        #  self |= other
```

7.4.5. Pozostałe wybrane metody działań

Metoda `__del__` klasy `object` jest `destruktorem`. Służy ona do niszczenia obiektów i dla obiektu klasy `Vector` jest wywoływana w następujący sposób: `del V`. Definicja destruktora dla klasy `Vector`, którego zadaniem jest zmniejszenie o 1 wartości zmiennej klasowej `_ile`, ma postać:

```
def __del__(self):
  Vector._ile -= 1
```

W przypadku klasy reprezentującej wektor będą potrzebne jeszcze dwa często wykorzystywane działania: `__getitem__` pobierająca wartość pola będącego listą znajdującą się pod wskazanym w argumencie indeksie oraz `__contains__`, której zadaniem jest sprawdzenie czy wartość podana jako argument jest elementem pola będącego listą.

```
def __getitem__(self, i):              # self[i]
  return self._vect[i]

def __contains__(self, element):       # element in self
  return element in self._vect
```

7.4.6. Przykład klasy `Vector`

Przedstawiamy definicję klasy `Vector` ze zdefiniowanymi wybranymi metodami tej klasy opisanymi we wcześniejszych podrozdziałach.

```
# vector.py
class Vector:
  _ile = 0

  def __init__(self, rozmiar = 0, lista = []):
    if (n:=len(lista)) != rozmiar:
      self._rozmiar =  n
    else:
      self._rozmiar = rozmiar
    self._vect = []
    for i in lista:
      self._vect.append(i)
    Vector._ile += 1
  def __del__(self):
    Vector._ile -= 1

  def get_ile(cls):
    return cls._ile

  def length(self):
    return self._rozmiar
  def __add__(self, other):
    if self._rozmiar != other._rozmiar:
      raise ValueError('Różne rozmiary wektorów!')
    V = Vector(self._rozmiar, self._vect)
    for i in range(self._rozmiar):
      V._vect[i] += other._vect[i]
    return V

  def __radd__(self, other):
    if isinstance(other,int):
      V = Vector(self._rozmiar)
      for i in range( self._rozmiar):
        V._vect.append(self._vect[i])
      for i in range(self._rozmiar):
        V._vect[i] += other
      return V
```

```
38      elif isinstance(other, list):
39        V = Vector(len(other), other)
40        return self + V
41      else:
42        return NotImplemented
43
44    def __iadd__(self, other):
45      if self._rozmiar == other._rozmiar:
46        for i in range(self._rozmiar):
47          self._vect[i] += other._vect[i]
48        return self
49      else:
50        raise ValueError('Różne rozmiary wektorów!')
51
52    def __mul__(self, alpha):
53      V = Vector(self._rozmiar)
54      print('mul=',self._rozmiar)
55      for i in range( self._rozmiar):
56        print(self._vect[i])
57        V._vect.append(self._vect[i])
58      for i in range(self._rozmiar):
59        V._vect[i] *= alpha
60      return V
61
62    def sum(self):
63      return sum(self._vect)
64    def __getitem__(self, i):
65      return self._vect[i]
66
67    def __ne__(self, other):
68      if self._rozmiar != other._rozmiar:
69        return True
70      for i in range(self._rozmiar):
71        if self._vect[i] != other._vect[i]:
72          return True
73      return False
74
75    def __contains__(self, element):
76      return element in self._vect
77
78    def __str__(self):
79      s = '['
```

```
    for  i in self._vect[:-1]:
      s += str(i) + ', '
    s += str(self._vect[-1]) + ']'
    return s
```

```
# main_Vector.py
from vector import Vector

def main():
  print('Liczba obiektów klasy Vector:', Vector._ile)
  V = Vector(5, [1,2,3,4,5])
  print('V =', V)
  V1 = Vector(4, [1,2,3,4])
  print('V1 =', V1)
  try:
    W = V + V1
  except ValueError as w:
    print(w)
  print('5 + V:', 5 + V)
  print('[1,1,1,1,1] + V:', [1,1,1,1,1] + V)
  try:
    print('2.2 + V:', 2.2 + V)
  except:
    print('UPS! Coś poszło nie tak!')
  V+=V
  print('V+=V:', V)
  V1 = Vector(5, [1,2,3,4,5])
  print('V1 =', V1)
  try:
    W = V + V1
  except ValueError as w:
    print(w)
  print('Po +:', W._rozmiar)
  print('Po +:', W._vect)
  W = W * 3
  print('W = W * 3:', W)
  print('suma W:', W.sum())
  print('W[2] =', W[2])
  #print('W[12] =', W[12])
  print('18 in W?', 18 in W)
  print('1 in W?', 1 in W)
  print('W != W?', W != W)
```

```
38   print('V =', V)
39   print('W =', W)
40   print('V != W?', V != W)
41   print('Liczba obiektów klasy Vector:', V.get_ile())
42   del V
43   print('Liczba obiektów klasy Vector:', Vector._ile)
44
45 if __name__ == "__main__":
46   main()
```

7.5. Właściwości

Python pozwala programistom na bezpośredni dostęp do zmiennych instancji, bez konieczności tworzenia oprzyrządowania w postaci `getterów` i `setterów`, powszechnych w innych językach obiektowych. Brak metod ustawiających i odczytujących wartości czyni kod klas w Pythonie czystszym i prostszym, ale w niektórych sytuacjach użycie `getterów` i `setterów` może być wygodne. Załóżmy, że chcielibyśmy dokonać jakiejś operacji na wartości, zanim przypiszemy ją do zmiennej instancyjnej. Albo przydałoby się obliczanie wartości zmiennej w „locie". W obu wypadkach metody typu `get` i `set` byłyby odpowiedzią na to zapotrzebowanie, ale ich kosztem byłaby utrata charakterystycznej dla Pythona łatwości dostępu do zmiennych instancji.

Rozwiązaniem jest tu używanie `właściwości`, która łączy możliwość dostępu do zmiennej instancji poprzez `gettery` lub `settery` oraz przejrzystość typowej dla Pythona notacji zmiennych instancji. Aby stworzyć właściwość, potrzebujemy dekoratora `@property` oraz metody typu `get` o nazwie takiej samej, jak właściwość. Bez funkcji udostępniającej ustawianie wartości, taka właściwość jest dostępna jedynie do odczytu. Aby móc ją zmienić, potrzebna jest metoda typu `set`.

LISTING 7.12: *Klasa* `NemeVector` *z użyciem właściwości*

```
1  # namevector.py
2  class NameVector(Vector):
3
4    def __init__(self, rozmiar, lista, nazwa):
5      Vector.__init__(self,rozmiar, lista)
6      self.__name = nazwa
7
8    @property
9    def name(self):
10     return self.__name
11
```

```
12   @name.setter
13   def name(self, new_name):
14     self.__name = new_name
15
16   def __str__(self):
17     s = self.__name + ' = ['
18     for  i in self._vect[:-1]:
19       s += str(i) + ', '
20     s += str(self._vect[-1]) + ']'
21     return s
```

Listing 7.13: *Użycie właściwości klasy* `NemeVector`

```
1  # main_namevector.py
2  from vector import Vector
3  from namevector import NameVector
4
5  def main():
6    nV = NameVector(5,[11,12,13,14,15],"wektorek")
7    print(nV)
8    print(nV.name)
9    nV.name = "Vec"
10   print(nV)
11   print(type(nV))
12   del nV
13 if __name__ == "__main__":
14   main()
```

W wyniku wykonania programu z listingu 7.13 na ekranie pojawią się informacje:

```
wektorek = [11, 12, 13, 14, 15]
wektorek
Vec = [11, 12, 13, 14, 15]
<class 'vector.NameVector'>
```

7.6. Serializowanie obiektów Pythona

Moduł `pickle` pozwala na serializację i deserializację obiektów Pythona. `Serializacja` (inaczej *marshalling*, *pickling* lub *flattening*) to proces przekształcania obiektu w ciąg bajtów. `Deserializacja` (inaczej *demarshalling* lub *unpicking*) to proces przeciwny - zamiana kolejnych bajtów na obiekt. Uzyskany w ten sposób ciąg bajtów można zapisać do pliku lub przesłać przez sieć. Dane zapisane w pliku mogą później posłużyć do odtworzenia stanu programu przy jego kolejnym uruchomieniu.

Funkcja `dumps` pozwala na serializację obiektu do ciągu bajtów lub inaczej pozwala zapisać obiekt w łańcuchu znaków. Ciąg bajtów można zapisać do pliku lub przesłać przez sieć.

```
from pickle import dumps

phone_book = {"Jonna": "542124", "Maciej": "542323"}

bytes = dumps(phone_book)
print(bytes)

"""output:
b'\x80\x04\x95(\x00\x00\x00\x00\x00\x00\x00}
  \x94(\x8c\x05Jonna\x94\x8c\x06542124\x94\x8c
  \x06Maciej\x94\x8c\x06542323\x94u.'
"""
```

Zapisanie danych do pliku możemy zrealizować za pomocą funkcji `dump`. Plik, w którym chcemy zapisać dane, musi zostać otwarty w trybie binarnym. Należy również pamiętać o jego zamknięciu.

```
from pickle import dump

phone_book = {"Jonna": "542124", "Maciej": "542323"}

with open('app_data.pickle', 'wb') as file:
    dump(phone_book, file)
```

Funkcja `loads` pozwala zamienić ciąg bajtów na obiekt (pozwala na odtwarzanie z łańcucha znaków).

```
from pickle import loads

bytes = (b'(dp0\nVJonna\np1\nV542124\np2\nsVMaciej\np3\nV542323\
    np4\ns.')

phone_book = loads(bytes)

print(phone_book)

"""output: {'Jonna': '542124', 'Maciej': '542323'} """
```

Odczyt danych z pliku możemy zrealizować za pomocą funkcji `load`. Plik, z którego chcemy odczytać dane, musi zostać otwarty w trybie binarnym. Należy również pamiętać o jego zamknięciu.

```
from pickle import load

with open('app_data.pickle', 'rb') as file:
    phone_book = load(file)
    print(phone_book)

"""output: {'Jonna': '542124', 'Maciej': '542323'} """
```

W większości programów funkcje `dump` i `load` wystarczą do skutecznego korzystania z modułu `pickle`. Rozwiązanie to działa dla większości typów danych Pythona i klas zdefiniowanych przez użytkowników. Jeżeli korzystamy z biblioteki, która umożliwia zapisywanie i odtwarzanie obiektów Pythona w bazach danych lub przesyłanie obiektów przez sieć, bardzo możliwe, że używa ona modułu `pickle`. Moduł `pickle` odpowiada za charakterystyczne dla Pythona samoopisowe kodowanie danych. Dzięki temu, że jest samoopisowe, serializowane dane zawierają informacje o początku i końcu każdego obiektu oraz o jego typie. Dlatego nie trzeba martwić się o definiowanie rekordów – kod działa i bez tego.

```
>>> import pickle
>>> f = open("somedata", "wb")
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump("Witaj", f)
>>> pickle.dump({"Jabłko", "Gruszka", "Banan"}, f)
>>> f.close()
>>> f = open("somedata", "rb")
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'Witaj'
>>> pickle.load(f)
{'Jabłko', 'Gruszka', 'Banan'}
>>>
```

W ten sposób można serializować funkcje, klasy i obiekty, przy czym w wygenerowanych danych zakodowane są tylko referencje do powiązanych obiektów. Oto przykład:

```
import math
import pickle
print(pickle.dumps(math.log))
print(pickle.dumps([math.sin, math.cos]))
```

W momencie deserializacji program przyjmuje, że cały potrzebny kod źródłowy jest dostępny. Moduły, klasy i funkcje są w razie potrzeby automatycznie importowane. Gdy dane Pythona są współużytkowane przez interpretery z różnych komputerów, może to

utrudniać konserwację kodu, ponieważ wszystkie komputery muszą mieć dostęp do tego samego kodu źródłowego. Funkcji `pickle.load` nigdy nie należy używać do niezaufanych danych. W ramach wczytywania kodu moduł `pickle` automatycznie pobiera moduły i na ich podstawie tworzy obiekty. Napastnik, który wie, jak działa moduł `pickle`, może przygotować specjalnie spreparowane dane powodujące, że Python wykona określone polecenia systemowe. Dlatego moduł `pickle` należy stosować tylko wewnętrznie w interpreterach, które potrafią uwierzytelniać siebie nawzajem.

```
# W_strumieniu danych znajduje się wywołanie funkcji
# system(), która uruchamia podane polecenie w konsoli.

import pickle
bytes = b"cos\nsystem\n(S'ls -la /'\ntR."
pickle.loads(bytes)

"""output
razem 2097236
drwxr-xr-x  19 root root  4096 mar 17  2022 .
drwxr-xr-x  19 root root  4096 mar 17  2022 ..
lrwxrwxrwx   1 root root     7 mar 17  2022 bin -> usr/bin
drwxr-xr-x   4 root root  4096 cze 18  2022 boot
drwxr-xr-x   2 root root  4096 mar 17  2022 cdrom
drwxr-xr-x  20 root root  4560 mar 25 18:01 dev
...
"""
```

Niektórych obiektów nie można zserializować w ten sposób. Są to zwykle obiekty mające zewnętrzny stan w systemie, takie jak otwarte pliki, otwarte połączenia sieciowe, wątki, procesy, ramki stosu itd. W klasach zdefiniowanych przez użytkownika można czasem obejść to ograniczenie, udostępniając metody `__getstate__` oraz `__setstate__`. Wtedy funkcja `pickle.dump` wywołuje metodę `__getstate__`, aby pobrać serializowany obiekt, a przy deserializacji wywoływana jest metoda `__setstate__`. Aby zilustrować możliwości tego podejścia, na następnym slajdzie przedstawiono klasę ze zdefiniowanym wewnętrznie wątkiem, którą można zarówno serializować, jak i deserializować.

```
import time
import threading

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
```

```
8           self.thr.daemon = True
9           self.thr.start()
10      def run(self):
11          while self.n > 0:
12              print('T-minus', self.n)
13              self.n -= 1
14              time.sleep(5)
15      def __getstate__(self):
16          return self.n
17      def __setstate__(self, n):
18          self.__init__(n)
```

```
>>> import pickle
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
>>> # Po pewnym czasie
>>> f = open("cstate.p", "wb")
>>> pickle.dump(c, f)
>>> f.close()
```

Teraz możemy wyjść z interpretera Pythona i po ponownym jego uruchomieniu wywołać następujący kod:

```
>>> import pickle
>>> f = open("cstate.p", "rb")
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

Widzimy, jak wątek w magiczny sposób ponownie zaczyna działać i wznawia pracę od miejsca, w którym zakończył ją w momencie serializowania.

Moduł `pickle` nie zapewnia wysokiej wydajności kodowania dużych struktur danych, np. tablic binarnych tworzonych przez takie biblioteki jak moduł `array` lub `numpy`. Jeżeli chcemy przenosić duże ilości danych tablicowych, lepszym rozwiązaniem może być zapisanie ich w pliku lub zastosowanie standardowego kodowania, np. `HDF5` (obsługiwanego przez niestandardowe biblioteki `https://docs.h5py.org/en/stable/`). Ponieważ

moduł `pickle` działa tylko w Pythonie i wymaga kodu źródłowego, zwykle nie należy go używać do długoterminowego przechowywania danych. Jeżeli kod źródłowy zostanie zmodyfikowany, wszystkie przechowywane dane mogą stać się nieczytelne.

Przy przechowywaniu danych w bazach danych lub archiwach zwykle lepiej jest stosować bardziej standardowe kodowania, np. `XML`, `CSV` lub `JSON` (`https://docs.python.org/3/library/json.html#module-json`). Są one w większym stopniu ustandaryzowane, obsługuje je wiele języków i są lepiej dostosowane do zmian w kodzie źródłowym. Ponadto warto pamiętać, że moduł `pickle` udostępnia wiele różnych opcji i ma skomplikowane przypadki brzegowe. Przy wykonywaniu typowych zadań nie trzeba się nimi przejmować. Jeżeli jednak pracujemy nad rozbudowaną aplikacją, która do serializacji używa modułu `pickle`, należy zapoznać się z jego oficjalną dokumentacją pod linkiem: `https://docs.python.org/3/library/pickle.html`.

Zachęcamy czytelnika do samodzielnego zapoznania się z innymi modułami Pythona służącymi do pracy z danymi przechowywanymi w plikach binarnych, np. takim, jak moduł `shelve` implementujący interfejs BSD bazy danych (ang. *Berkeley Software Distribution Database Interface* (`https://docs.python.org/3/library/shelve.html`). Jest to prosty w użyciu moduł do przechowywania i odczytywania danych z pliku na dysku, idealny do mniej skomplikowanych aplikacji, które potrzebują łatwego sposobu na trwałe przechowywanie struktur danych Pythona.

Ze względu na wykorzystywanie modułu `pickle`, często w sposób automatyczny, w innych modułach służących pracy z danymi zapisanymi na dysku, co ma miejsce między innymi we wspomnianym wcześniej module `shleve`, ograniczyliśmy się w prezentowanym podręczniku do opisu tylko tego modułu.

7.7. Zadania do samodzielnego rozwiązania

1. Zaimplementuj klasę `Adres` do przechowywania informacji o: numerze domu, ulicy, opcjonalnie numerze mieszkania, mieście i kodzie pocztowym. Zdefiniuj inicjalizator tak, aby obiekt mógł zostać utworzony na jeden z dwóch sposobów: z numerem mieszkania lub bez niego. Dostarcz metodę `show`, która wypisuje adres z ulicą w jednym wierszu oraz kodem pocztowym i miastem w następnej linii.
   Dostarcz metodę `comesBefore(self, other)`, która sprawdza, czy dany adres jest przed innym, gdy porównujemy go według kodu pocztowego.
2. Zaimplementuj klasę `Car` o następujących właściwościach. Samochód ma określoną wydajność paliwową (mierzoną w kilometrach/litr) i pewną maksymalną ilość paliwa w zbiorniku. Wydajność jest określona w inicjalizatorze, a początkowy poziom paliwa wynosi 0. Dostarcz metodę `drive`, która symuluje jazdę samochodem przez pewien dystans, zmniejszając poziom paliwa w zbiorniku, metodę `getFuelLevel`,

która zwraca aktualny poziom paliwa, oraz metodę `addFuel`, która symuluje tankowanie, przy czym nie można przekroczyć maksymalnej pojemności baku. Przykładowe użycie:

```
my_car = Car(20, 40) # Wydajność 20 km/litr, pojemność baku 40
my_car.addFuel(30)   # Zatankuj co najwyżej 30 litrów
my_car.drive(100)    # Przejedź 100 m
print(my_car.getFuelLevel()) # Wydrukuj ilość pozostałego
    paliwa
```

3. Zaimplementuj klasę `Student`. Na potrzeby tego ćwiczenia student ma imię i nazwisko oraz całkowity wynik quizów. Zdefiniuj odpowiedni inicjalizator oraz metody:
   - `getName()` - zwracającej nazwisko studenta,
   - `addQuiz(score)` - dodającą wynik quizu,
   - `getTotalScore()` - zwracającą całkowity wynik quizów oraz
   - `getAverageScore()` - zwracającą średni wynik quizów; aby w ostatniej z tych metod obliczyć średni wynik ze wszystkich quizów, należy przechowywać w odpowiednim polu liczbę quizów, do których przystąpił student.
4. Utwórz plik `pies.py`, w którym zdefiniuj i przetestuj klasę `Pies` reprezentującą psa, który ma:
   - `imie` - domyślnie `'Rex'`,
   - `zabawki_lubiane` - swoje ulubione zabawki - domyślnie lista z jedynym elementem `kość`, elementy na tej liście nie powtarzają się,
   - `zabawki_nielubiane` - zabawki, których pies nie lubi - domyślnie lista z jedynym elementem `pies`, elementy na tej liście nie powtarzają się,

   a ponadto pies potrafi:
   - przywitać się i przedstawić,
   - powiedzieć, jakie są jego lubiane/nielubiane zabawki i ile ich jest,
   - polubić jakąś zabawkę, np. nową lub dotąd nielubianą,
   - przestać lubić jakąś zabawkę.

   Nowo napotkaną zabawkę pies może tylko polubić lub przejść obok niej obojętnie. Zastanów się w jaki sposób zapamiętać nową zabawkę, wobec której pies przejdzie obojętnie. Ważne: listy lubianych i nielubianych zabawek nie mają elementów wspólnych.
5. Korzystając z klasy `Pies`, stwórz symulację wędrówki psa po okręgu, który po drodze powinien mijać dużo pudełek z losowymi, ale jednego rodzaju zabawkami w jednym pudełku. Dodatkowo zakładamy, że zawartość pudełek może się powtarzać. Kiedy pies napotyka pudełko, widzi trzymane tam zabawki i może:
   - przejść obok pudełka obojętnie,
   - losowo polubić jedną z zabawek lub wybrać sobie tę, która mu się spodoba,

- przestać lubić wybraną zabawkę, o ile wcześniej była jego ulubioną.

Pies dotąd wyrusza w wędrówkę, aż osiągnie zadaną z góry liczbę powtórzeń (np. `100`) lub co najmniej zawartość jednego pudełka wyczerpie się. Po zakończeniu wędrówki pies pokazuje zabawki, które lubi i których nie lubi. Dla uproszczenia, zakładamy że pies w każdej iteracji napotyka losowe pudełka. Listy zabawek (pudełka) są na początku wczytywane z pliku, który samemu trzeba stworzyć i zapisać w osobnych wierszach rozdzielone spacją nazwy zabawek jednego rodzaju.

6. Rozszerz definicję omawianej w podrozdziale 7.4.6 klasy `Vector` o definicje przeciążające te operatory, które niezbędne są do jej prawidłowego funkcjonowania.
7. Napisz klasę `Wymierna` reprezentującą liczby wymierne `p/q`. Liczby `p` i `q` powinny być pamiętane jako względnie pierwsze z dodatnim `q`. Zaimplementuj:
   (a) Inicjalizator z dwoma argumentami całkowitymi, licznik i mianownik, przy czym domyślną wartością licznika powinno być zero, a mianownika jeden. Inicjalizator powinien działać poprawnie również jeżeli podane argumenty nie są względnie pierwsze lub mianownik jest ujemny.
   (b) Funkcje składowe `getLicznik` i `getMianownik` zwracające odpowiednio licznik i mianownik liczby.
   (c) Funkcję składową `__repr__` zwracającą łańcuch znaków reprezentujący liczbę wymierną.
   (d) Funkcję składową `__float__` zwracającą wartość typu `float` odpowiadającą danej liczbie wymiernej.
   (e) Funkcje składowe `__add__` oraz `__sub__`.
   (f) Funkcje składowe `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`.

   W funkcji `main` wczytaj licznik i mianownik dla dwóch liczby wymiernych, utwórz z wczytanych liczb dwie liczby wymierne, a następnie wypisz w kolejnych liniach wyniki uzyskane z zastosowania zdefiniowanych operatorów.
8. Rozszerz definicję klasy z poprzedniego zadania poprzez zdefiniowanie funkcji składowych `__mul__` oraz `__truediv__`.
9. Rozszerz definicję klasy z poprzedniego zadania poprzez zdefiniowanie funkcji implementującej jednoargumentowy minus służący do zmiany wartości na przeciwną.
10. Zaimplementuj funkcję `__eq__` w sposób wykorzystujący fakt, iż dwie liczby są równe wtedy i tylko wtedy, gdy żadna z nich nie jest mniejsza od drugiej.
11. Wykorzystując atrybut `__slots__` zaimplementuj w pliku `punkt.py` klasę `Punkt` z właściwościami `__x` oraz `__y`. Dla obu własności zdefiniuj `setter` oraz `deleter`. Zdefiniuj także metody specjalne `__repr__` i `__str__` w sposób jaki został zaprezentowany w rozdziale. Przetestuj klasę `Punkt` w funkcji `main` zdefiniowanej w pliku `test_punkt.py`.

12. Wykorzystując atrybut `__slots__` zaimplementuj klasę `NazwanyPunkt` dziedziczącą po klasie `Punkt`. Przetestuj klasę `NazwanyPunkt` w funkcji `main` zdefiniowanej w pliku `test_nazwanypunkt.py`.
13. Napisz program, w którym korzystając z klas `Punkt` oraz `NazwanyPunkt` z wcześniejszych zadań utwórzysz listę `punkty` z czterema obiektami tych klas (po dwa obiekty z każdej klasy). Stosując moduł `pickle` zapisz listę `punkty` w pliku `punkty.pkl`.
14. W nowo posadzonym lesie rosną drzewa. Każde drzewo jest zasadzone przez człowieka i początkowo ma średnicę 1 metra. Nagle w lesie zaczyna grasować bóbr, który codziennie wyrusza na łowy i podgryza losowe drzewo. Wszystkie pozostałe drzewa codziennie trochę rosną - ich średnica zwiększa się o `1%`. Podgryzane drzewo zmniejsza swoją średnicę aż o `0,2 m` i może rosnąć dopiero kolejnego dnia. Bóbr, który początkowo ma siłę równą `2` męczy się przy gryzieniu drzew, a jego siła zmniejsza się o wartość równą `7%` długości średnicy drzewa w metrach. Sprawdź, po ilu dniach albo bóbr wyczerpie swoje siły albo powali drzewo, gdy jego średnica zmniejszy się do zera. Zdefiniuj klasy `Bobr`, `Drzewo` i `Las` oraz funkcję główną `main` symulującą opisane wyżej działania.

Rozdział 8

Zaawansowane elementy języka Python

Rozdział ten poświęcony jest zaawansowanym technikom pracy z danymi w języku Python, które pozwalają na efektywne zarządzanie i manipulowanie kolekcjami oraz sekwencjami danych. Zapoznamy się z listami składanymi, iteratorami, generatorami, typami wyliczeniowymi oraz sposobami ich wykorzystania do usprawnienia działania programów, jak również poprawy czytelności kodu i zmniejszenia jego objętości.

8.1. Listy składane

`Listy składane` (ang. *list comprehension*) to konstrukcja, która pozwala na zwięzłe tworzenie list. Typowym zastosowaniem jest utworzenie listy, której elementy są wynikiem zastosowania pewnej operacji do każdego elementu innej sekwencji lub obiektu iterowalnego. Innym typowym zastosowaniem jest utworzenie listy z tych elementów sekwencji, które spełniają pewne warunki. Listę składaną tworzy się w nawiasach kwadratowych, w których występuje wyrażenie, a po nim klauzula `for`, a następnie pewna ilość klauzul `for` lub `if`, przy czym te dodatkowe klauzule nie muszą wystąpić.

Przedstawimy teraz kilka przykładów tworzenia list składanych:

1. lista kwadratów 10 początkowych liczb naturalnych;
   ```
   kwadraty = [x**2 for x in range(1,11)]
   ```
2. lista znaków łańcucha z pominięciem spacji i znaków cyfr;
   ```
   znaki = [c for c in napis if not (c.isdigit() or c.isspace())]
   ```
3. lista wszystkich par (krotek) liczb, w których pierwszy element pary wybieramy z listy `[1,2,3]`, drugi z listy `[3,1,4]`, przy czym elementy pary są różne.
   ```
   pary = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
   ```

Na listingu 8.1 tworzone są następujące listy składane:
linia 3: podwojonych wartości sekwencji `wektor`;
linia 5: tylko dodatnich wartości sekwencji `wektor`;
linia 7: wartości bezwzględnych z wartości sekwencji `wektor`;
linia 10: napisów sekwencji `koszyk` ze znakami zamienionymi na duże litery;
linia 12: par składających się z liczb z zakresu `range(1,6)` oraz ich kwadratów;
linia 15: spłaszczenia macierzy `wektor` do sekwencji jednego wymiaru jej elementów.

LISTING 8.1: *Przykłady tworzenia list składanych*

```
1  def main():
2      wektor = [-4, -2, 0, 2, 4]
3      podwojone =[x * 2 for x in wektor]
4      print(podwojone)
5      tylko_dodatnie =  [x for x in wektor if x > 0]
6      print(tylko_dodatnie)
7      wartosci_bezwzgledne = [abs(x) for x in wektor]
8      print(wartosci_bezwzgledne)
9      koszyk = ['jabłko','gruszka','śliwka','mango','banan']
10     koszyk = [owoc.upper() for owoc in koszyk]
11     print(koszyk)
12     pary = [(x, x**2) for x in range(1,6)]
13     print(pary)
14     wektor = [[1,2,3], [4,5,6], [7,8,9]]
15     wektor = [num for elem in wektor for num in elem]
16     print(wektor)
17
18 if __name__=='__main__':
19     main()
```

W wyniku uruchomienia programu z listingu 8.1 otrzymamy na ekranie:

```
[-8, -4, 0, 4, 8]
[2, 4]
[4, 2, 0, 2, 4]
['JABŁKO', 'GRUSZKA', 'ŚLIWKA', 'MANGO', 'BANAN']
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Jeśli w liście składanej chcemy użyć instrukcji warunkowej `if` z klauzulą `else`, to musimy ją umieścić przed pętlą `for`. Na listingu 8.2 przedstawiamy przykład z zastosowaniem powyższej konstrukcji do utworzenia listy potrojonych dodatnich wartości sekwencji `wektor` oraz kwadratów jej elementów dla wartości ujemnych lub zera.

Listing 8.2: *Lista składana z konstrukcją* `if-else-for`

```
1 def main():
2     wektor = [-4, -2, 0, 2, 4]
3     if_else_for = [x*3 if x > 0 else x**2 for x in wektor]
4     print(if_else_for)
5 if __name__=='__main__':
6     main()
```

Początkowe wyrażenie w liście składanej może być dowolnym wyrażeniem, w tym zawierającym inną listę składaną. Na listingu 8.3 przedstawiamy program, w którym stosujemy listy zagnieżdżone do konstrukcji macierzy transponowanej (linia 7) do macierzy `matrix`.

Listing 8.3: *Tworzenie macierzy transponowanej*

```
1 def main():
2     matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
3     for i in matrix:
4         for j in i:
5             print("{0:4}".format(j),end="")
6         print()
7     A = [[row[col] for row in matrix] for col in range(4)]
8     print()
9     for i in A:
10        for j in i:
11            print("{0:4}".format(j),end="")
12        print()
13 if __name__=='__main__':
14     main()
```

Zwracamy uwagę, aby wszędzie tam, gdzie jest to możliwe, używać odpowiedniej funkcji wbudowanej zamiast tworzyć skomplikowane konstrukcje list zagnieżdżonych.

Jedną z takich funkcji wbudowanych jest funkcja `zip`, która pobiera odpowiadające sobie elementy z jednego lub więcej obiektów iterowalnych i tworzy z nich krotki aż do wyczerpania najkrótszego obiektu iterowalnego. Przy czym jeśli chcemy, aby powstała struktura typu `zip object` była listą, musimy zastosować funkcję `list` dla utworzonej przez funkcję `zip` struktury danych.

Dla przykładu rozważmy dwie listy: `liderzy`, gdzie zapisane są imiona i nazwiska osób znanych, liderów branży IT oraz `id` zawierającą identyfikatory całkowite z zakresu od 1 do 4. Stosując funkcję `zip` względem tych sekwencji otrzymujemy z użyciem funkcji `list` listę par powiązanych ze sobą informacji.

```
>>> id = [1, 2, 3, 4]
>>> liderzy = ['Elon Musk','Tim Cook','Bill Gates','Yang Zhou']
>>> plec = ['male','male','male','male']
>>> record = zip(id, liderzy)
>>> list(record)
[(1,'Elon Musk'),(2,'Tim Cook'),(3,'Bill Gates'),(4,'Yang Zhou')]
```

Wywołanie funkcji `zip` tylko dla jednego zbioru utworzy tyle jednoelementowych krotek ile jest elementów w sekwencji `id`.

```
>>> record = zip(id)
>>> list(record)
[(1,), (2,), (3,), (4,)]
```

Użycie bezargumentowej funkcji `zip` skutkuje utworzeniem zbioru pustego, a w konsekwencji z użyciem funkcji `list` daje pustą sekwencję.

```
>>> record = zip()
>>> list(record)
[]
```

Funkcję `zip` można stosować dla więcej niż dwóch argumentów. Tu prezentujemy jej wywołanie dla trzech zbiorów danych: `id`, `liderzy` i `plec`. W wyniku otrzymujemy trójelementowe krotki.

```
>>> id = [1, 2, 3, 4]
>>> liderzy = ['Elon Musk','Tim Cook','Bill Gates','Yang Zhou']
>>> plec = ['male','male','male','male']
>>> record = zip(id, liderzy,plec)
>>> list(record)
[(1, 'Elon Musk', 'male'), (2, 'Tim Cook', 'male'),
(3, 'Bill Gates', 'male'), (4, 'Yang Zhou', 'male')]
```

Możemy także wykorzystać funkcję `zip` do operacji rozpakowywania. Użycie `*` - asteryska względem obiektu `record` spowodowało rozpakowanie krotek z listy jako czterech odrębnych krotek:
`(1,'Elon Musk') (2,'Tim Cook') (3,'Bill Gates') (4,'Yang Zhou')`,
aby następnie funkcja `zip` połączyła je w odrębne krotki identyfikatorów `id` i liderów `liderzy`.

```
>>> record = [(1,'Elon Musk'), (2,'Tim Cook'), (3,'Bill Gates'),\
            (4,'Yang Zhou')]
>>> id, liderzy = zip(*record)
```

```
>>> id
(1, 2, 3, 4)
>>> liderzy
('Elon Musk', 'Tim Cook', 'Bill Gates', 'Yang Zhou')

>>> A = list(zip(*record))
>>> A
[(1, 2, 3, 4), ('Elon Musk', 'Tim Cook', 'Bill Gates', 'Yang Zhou')]
```

A teraz pokażemy jak prosto można otrzymać macierz transponowaną używając funkcji `zip`.

```
>>> matrix = [[1, 2, 3], [1, 2, 3]]
>>> matrix_T = [list(i) for i in zip(*matrix)]
>>> matrix_T
[[1, 1], [2, 2], [3, 3]]
```

Listy składane mają zwięzły zapis i są często spotykanym wzorcem kodu budowania list wyników w Pythonie. W zależności od wersji Pythona i samego kodu programu, listy składane mogą działać o wiele szybciej od ręcznie zapisanych instrukcji pętli `for` (często nawet dwukrotnie szybciej). Iteracje list składanych wykonywane są wewnątrz interpretera z szybkością języka C, a nie z szybkością kodu Pythona. Z tego powodu używanie ich może być korzystne z punktu widzenia wydajności.

Listę składaną możemy wykorzystać również do operacji na plikach. Jeżeli otworzymy plik w wyrażeniu, lista składana automatycznie wczyta po jednym wierszu z pliku na raz i doda go do listy wyników. Dzięki zastosowaniu list składanych osiągamy bardziej wydajne i szybsze rozwiązanie. Listy składane automatycznie zamykają plik, gdy tymczasowy obiekt jest czyszczony po wykonaniu wyrażenia.

```
>>> lines = [line for line in open('wierszyk.txt')]
>>> lines
['Eeny, meeny, miny, moe,\n', 'Catch a tiger by the toe.\n',
'If he hollers, let him go,\n', 'Eeny, meeny, miny, moe.\n']
```

Pętla `for` zagnieżdżona w wyrażeniu listy składanej może mieć powiązaną klauzulę `if`, pozwalającą na „odfiltrowanie" tych elementów wyników, dla których test nie jest prawdziwy.

```
>>> file = open('wierszyk.txt')
>>> lines = [line.rstrip() for line in file if line[0] == 'E']
>>> lines
['Eeny, meeny, miny, moe,', 'Eeny, meeny, miny, moe.']
```

8.2. Funkcje anonimowe

Obok instrukcji `def` Python udostępnia również formę wyrażenia generującego obiekty funkcji nazywanego `wyrażeniem lambda` (lub krócej po prostu `lambda`). Nazwa `lambda` została zapożyczona z języka `LISP`, który z kolei zaczerpnął ją z rachunku lambda (Alonzo Church; lata trzydzieste XX wieku; teza Churcha-Turinga), będącego formą logiki symbolicznej. W Pythonie jest to tylko słowo kluczowe wprowadzające odpowiednią składnię wyrażenia. Tak jak `def` wyrażenie lambda tworzy funkcję, którą można wywołać później, jednak zwraca tę funkcję, zamiast przypisywać ją do nazwy. Z tego powodu wyrażenia lambda nazywane są czasami `funkcjami anonimowymi` (nienazwanymi). W praktyce często wykorzystywane są jako sposób skrótowego zapisania definicji funkcji lub opóźnienia wykonania fragmentu kodu.

Na ogólną formę wyrażenia lambda składa się słowo kluczowe `lambda`, po którym następuje dowolna liczba argumentów, a po nich, po dwukropku, wyrażenie:

```
1 lambda arg1, arg2, ..., argN : wyrażenie
```

Obiekty funkcji zwracane przez wykonanie wyrażeń lambda działają dokładnie tak samo jak te utworzone i przypisane przez instrukcję `def`. Wyrażenia lambda różnią się tym, że:

- `lambda` nie jest instrukcją, a wyrażeniem, dlatego może się pojawiać w miejscach, w których użycie `def` w składni Pythona nie jest dozwolone, np. wewnątrz literału listy czy w wywołaniu funkcji;
- ciałem `lambda` jest pojedyncze wyrażenie, a nie blok instrukcji i służy zapisywaniu prostych funkcji.

Przedstawiamy kilka przykładowych wyrażeń lambda, które:

- przyjmuje dwa parametry `x` oraz `y` i zwraca ich sumę;

  ```
  lambda x, y: x + y
  ```

- przyjmuje jeden parametr i zwraca wartość tego parametru powiększoną o 1;

  ```
  lambda x : x + 1
  ```

- jak wyżej, ale zamiast nazwy parametru podajemy znak podkreślenia `_`;

  ```
  lambda _ : _ + 1
  ```

- przypisujemy wyrażenie lambda do zmiennej i ją wykonujemy;

  ```
  zmienna = lambda x,y: x+y
  zmienna(2,3)
  ```

- samo wywołanie wyrażenia lambda.

  ```
  (lambda x,y: x+y)(3,4)
  ```

Na listingu 8.4 przedstawiamy program z użyciem jako klucza porównywania w wyszukiwaniu i sortowaniu zdefiniowanej przez użytkownika funkcji `year`.

Listing 8.4: *Użycie funkcji zwracającej klucz do porównywania*

```
1  def main():
2    d = {"Eve": 1999, "Ann": 2001, "Cay": 2000, "Bob": 2003}
3    print('max:', max(d.items()))
4    print('max by year:', max(d.items(), key=year))
5    print('sorted:', sorted(d.items()))
6    print('sorted by year:', sorted(d.items(), key=year))
7
8  def year(item):
9    return item[1]
10
11 if __name__ == '__main__':
12   main()
```

W wyniku wykonania programu z listingu 8.4 na ekranie pojawi się: maksymalny element znajdowany po kluczach słownika (linia 3), maksymalny element, ale szukany po wartościach słownika, czyli roku (linia 4), lista krotek posortowanych elementów słownika według jego kluczy (linia 5) oraz lista krotek posortowanych elementów słownika według jego wartości, czyli roku (linia 6).

```
OUTPUT:
max: ('Eve', 1999)
max by year: ('Bob', 2003)
sorted: [('Ann', 2001), ('Bob', 2003), ('Cay', 2000), ('Eve', 1999)]
sorted by year: [('Eve', 1999), ('Cay', 2000), ('Ann', 2001),
('Bob', 2003)]
```

Przypominamy, że funkcja `sorted` tworzy listę krotek odpowiadających elementom słownika posortowaną według klucza porównania, którym domyślnie jest operator `<` stosowany względem kluczy słownika.

Wyrażenie lambda możemy wykorzystać jako klucz do porównywania w funkcji sortowania (`sorted`) lub wyszukiwania elementu maksymalnego (`max`), np. elementów słownika, co pokazano na listingu 8.5.

Listing 8.5: *Użycie wyrażenia lambda zwracającego klucz do porównywania*

```
1 d = {"Eve": 1999, "Ann": 2001, "Cay": 2000, "Bob": 2003}
2 print('max:',max(d.items()))
3 print('max by year:',max(d.items(), key=lambda t: t[1]))
4 print('sorted:',sorted(d.items()))
5 print('sorted by year:',sorted(d.items(), key=lambda t: t[1]))
```

Wyniki wykonania programu z listingu 8.5 potwierdzają równoważność zastosowanych rozwiązań.

```
OUTPUT:
max: ('Eve', 1999)
max by year: ('Bob', 2003)
sorted: [('Ann', 2001), ('Bob', 2003), ('Cay', 2000), ('Eve', 1999)]
sorted by year: [('Eve', 1999), ('Cay', 2000), ('Ann', 2001),
('Bob', 2003)]
```

I jeszcze kilka przykładów zastosowania wyrażeń lambda w funkcjach sortowania:

- z kluczem porównywania wyznaczonym przez wyrażenie lambda odwołujące się do każdego klucza w słowniku `d`:

```
>>> d = {'b': 42, 'c': 1, 'a': 2}
>>> sorted(d.items())
[('a', 2), ('b', 42), ('c', 1)]
>>> sorted(d.items(), key=lambda t: t[0])
[('a', 2), ('b', 42), ('c', 1)]
>>> sorted(d.items(), key=lambda t: t[0], reverse=True)
[('c', 1), ('b', 42), ('a', 2)]
```

- z kluczem porównywania wyznaczonym przez wyrażenie lambda odwołujące się do każdej wartości w słowniku `d`:

```
>>> d = {'b': 42, 'c': 1, 'a': 2}
>>> sorted(d.items(), key=lambda t: t[1])
[('c', 1), ('a', 2), ('b', 42)]
>>> sorted(d.items(), key=lambda t: t[1], reverse=True)
[('b', 42), ('a', 2), ('c', 1)]
>>> ob = sorted(d.items(), key=lambda t: t[1], reverse=True)
```

W wyrażeniach `lambda` dozwolone jest stosowanie wyrażenia warunkowego postaci:

```
lambda arg1, arg2, ..., argN:
       wartość if wyrażenie_prawdziwe [else [...]]
```

W zależności od wartości `x` na ekran kierowany jest odpowiedni komunikat.

```
1 kom1 = "przyjdę dziś"
2 kom2 = "nie przyjdę dziś"
3
4 print((lambda x: kom1 if x > 0 and x < 10 else kom2)(2))
5 # przyjdę dziś
6
7 print((lambda x: kom1 if x > 0 and x < 10 else kom2)(12))
8 # nie przyjdę dziś
```

Funkcja wyższego rzędu jest to zwykła funkcja, z tą różnicą, że przyjmuje jako argument inną funkcję lub zwraca funkcję, np.

```
def funkcja(f, liczba):
  return f(liczba)
```

Pierwszym argumentem jest funkcja `f`, natomiast drugim `liczba`. Wywołanie funkcji wyższego rzędu z użyciem funkcji obliczającej pierwiastek kwadratowy z liczby 2 wygląda tak:

```
from math import sqrt
print(funkcja(math.sqrt, 2))
```

Wyłącznie na potrzeby wywołania funkcji wyższego rzędu można zdefiniować zwykłą funkcję:

```
def funkcja(f, liczba):
  return f(liczba)

def cube(x):
  return x * x * x

print(funkcja(cube, 2))
```

Z kolei stosując wyrażenie lambda, zapis ten można skrócić:

```
print(funkcja(lambda x: x * x * x, 2))
```

8.3. Typy wyliczeniowe

`Wyliczenie` to zbiór nazw symbolicznych powiązanych z unikalnymi, stałymi wartościami. W ramach wyliczenia elementy należące do niego mogą być porównywane według tożsamości, a samo wyliczenie może być powtarzane. Ponieważ wyliczenia są używane do reprezentowania stałych, zaleca się używanie nazw pisanych WIELKIMI LITERAMI dla elementów wyliczenia. W Pythonie moduł `enum` definiuje cztery klasy wyliczenia, których można użyć do zdefiniowania unikalnych zestawów nazw i wartości:

- `Enum` - klasa bazowa do tworzenia stałych wyliczanych;
- `IntEnum` - klasa bazowa do tworzenia stałych wyliczanych, które są również podklasami `int`;
- `Flag` - klasa bazowa do tworzenia wyliczanych stałych, które można łączyć za pomocą operatorów bitowych bez utraty ich przynależności do klasy `Flag`;
- `IntFlag` - klasa bazowa do tworzenia wyliczanych stałych, które można łączyć przy użyciu operatorów bitowych bez utraty ich przynależności do `IntFlag`; elementy klasy dziedziczącej po klasie `IntFlag` są również instancjami klasy `int`.

8.3.1. Klasa `Enum`

Wyliczenia są tworzone przy użyciu składni klas. Aby zdefiniować wyliczenie, należy w następujący sposób utworzyć podklasę `Enum`:

```
from enum import Enum
class Color(Enum):
  RED = 1
  GREEN = 2
  BLUE = 3
```

Elementy wyliczenia mają czytelną reprezentację łańcuchową, np:

```
>>> Color.RED
<Color.RED: 1>
```

Typem elementu wyliczenia jest wyliczenie, do którego ten element należy:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
```

Elementy wyliczenia mają atrybut, który zawiera tylko nazwę elementu:

```
>>> Color.RED.name
'RED'
```

Wyliczenia obsługują iterację, w kolejności zgodnej z definicją, np.

```
>>> for _ in Color:
...  print(_)
...
Color.RED
Color.GREEN
Color.BLUE
```

Czasami przydatne jest uzyskanie programowego dostępu do elementów wyliczeniowych, np. w sytuacji, gdy nie można użyć `Color.RED`, ponieważ dokładny kolor nie jest znany w momencie pisania programu.

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

Aby uzyskać dostęp do elementów wyliczenia po nazwie, należy użyć dostępu do elementu, np.

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

Jeżeli mamy element `enum` i potrzebujemy jego nazwy lub wartości możemy odwołać się do jego odpowiednich atrybutów, np.

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

Posiadanie dwóch elementów wyliczenia o tej samej nazwie jest niepoprawne, co jest zgłaszane w postaci `TypeError`, np.

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

Dwa elementy `enum` mogą mieć tę samą wartość. Jeżeli dwa elementy `A` i `B` mają tę samą wartość, przy czym `A` jest zdefiniowane jako pierwsze, to `B` jest aliasem `A`. Poszukiwanie po wartości odpowiadającej `A` i `B` zwróci `A`. Poszukiwanie po nazwie `B` również zwróci `A`:

```
>>> from enum import Enum
>>> class Shape(Enum):
...   SQUARE = 2
...   DIAMOND = 1
...   CIRCLE = 3
...   ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
```

```
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

W module `enum` jest zdefiniowana funkcja będąca dekoratorem klas wyliczeń (`@unique`). Przeszukuje ona atrybut `__members__` danego wyliczenia i wyszukuje wszystkie aliasy, które znajdzie. Jeżeli jakieś zostaną znalezione, zostanie zgłoszony wyjątek `ValueError` wraz ze szczegółami.

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...   ONE = 1
...   TWO = 2
...   THREE = 3
...   FOUR = 3
...
Traceback (most recent call last): ...
ValueError:
duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

Jeśli dokładna wartość jest nieistotna, można użyć obiektu klasy `enum.auto`, która użyta po raz pierwszy zwróci domyślną wartość 1, a przy każdym następnym odwołaniu zwróci wartość o 1 większą od poprzedniej.

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...   RED = auto()
...   BLUE = auto()
...   GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Wartości wybiera funkcja `_generate_next_value_()`, która może zostać nadpisana i dostosowana do wymagań użytkownika, np. przypisując tworzonym elementom wyliczenia jako wartości ich nazwy zamiast liczb naturalnych.

```
>>> from enum import Enum, auto
>>>
```

```
>>> class AutoName(Enum):
...   def _generate_next_value_(name, start, count, last_values):
...     return name
...
>>> class Ordinal(AutoName):
...   RED = auto()
...   GREEN = auto()
...
>>> list(Ordinal)
[<Ordinal.RED: 'RED'>, <Ordinal.GREEN: 'GREEN'>]
```

Iteracja po elementach wyliczenia nie dostarcza aliasów, np. dla zdefiniowanego wcześniej wyliczenia o nazwie `Shape` użycie pętli for nie pokaże aliasów.

```
>>> for element in Shape:
...   print(repr(element))
<Shape.SQUARE: 2>
<Shape.DIAMOND: 1>
<Shape.CIRCLE: 3>
```

Atrybut specjalny `__members__` jest uporządkowanym, przeznaczonym tylko do odczytu odwzorowaniem nazw na elementy. Zawiera on wszystkie nazwy zdefiniowane w wyliczeniu, w tym również aliasy.

```
>>> for name, member in Shape.__members__.items():
...   print("({},{})".format(repr(name), repr(member)))
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

Atrybut `__members__` może być użyty do szczegółowego, programowego dostępu do elementów wyliczenia, np. do znalezienia wszystkich aliasów wyliczenia `Shape`.

```
>>> L = [n for n, m in Shape.__members__.items() if m.name != n]
>>> L
['ALIAS_FOR_SQUARE']
```

Elementy wyliczenia są porównywane według tożsamości:

```
>>> Color.RED is Color.RED
True
```

```
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Porównania porządkowe między wartościami wyliczenia nie są obsługiwane, ponieważ elementy wyliczenia nie są liczbami całkowitymi.

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError:'<' not supported between instances of 'Color' and 'Color'
```

Porównania równości są jednak zdefiniowane.

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Porównania z wartościami niewyliczeniowymi zawsze będą zwracać wartość `False`:

```
>>> Color.BLUE == 2
False
```

Klasa `Enum` jest wywoływalna i udostępnia interfejs API (ang. *Application Programming Interface*), który przypomina semantykę klasy `namedtuple` z modułu `collections`. Pierwszym argumentem wywołania funkcji `Enum` jest nazwa wyliczenia; drugim argumentem jest źródło nazw elementów wyliczenia. Może to być łańcuch nazw oddzielonych białymi znakami, sekwencja nazw, sekwencja dwuelementowych krotek z parami `klucz-wartość` lub odwzorowanie (np. słownik) nazw na wartości. Dwie ostatnie opcje umożliwiają przypisanie wyliczeniom dowolnych wartości. Pozostałe automatycznie przypisują rosnące liczby całkowite, począwszy od 1. Aby określić inną wartość początkową, należy użyć argumentu start.

```
>>> Animal = Enum("Animal", "ANT BEE CAT DOG")
>>> Animal
<enum 'Animal'>
>>> repr(Animal.ANT )
'<Animal.ANT: 1>'
```

```
>>> Animal.ANT.value
1
>>> for animal in Animal:
...   print(repr(animal))
...
<Animal.ANT: 1>
<Animal.BEE: 2>
<Animal.CAT: 3>
<Animal.DOG: 4>
```

Zwracana jest nowa klasa wywodząca się z `Enum`. Innymi słowy, uprzednie przypisanie do zmiennej `Animal` jest równoważne z poniższym:

```
>>> class Animal(Enum):
...   ANT = 1
...   BEE = 2
...   CAT = 3
...   DOG = 4
...
```

Powodem domyślnego ustawiania 1 jako liczby początkowej, a nie 0, jest to, że 0 ma wartość `False`, natomiast wszystkie elementy wyliczenia przyjmują tylko wartość `True`.

8.3.2. Klasa IntEnum

Elementy klasy `IntEnum`, będącej jednocześnie podklasą klasy `Enum` oraz `int`, mogą być porównywane z liczbami całkowitymi. Ponadto elementy różnych podklas klasy `IntEnum` mogą być porównywane między sobą.

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...   CIRCLE = 1; SQUARE = 2
...
>>> class Request(IntEnum):
...   POST = 1; GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

Jednak nadal nie można podklasy `Shape` klasy `IntEnum` porównać do standardowych wyliczeń `Enum`.

```
>>> class Color(Enum):
...   RED = 1
...   GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

Wartości `IntEnum` zachowują się jak liczby całkowite w sposób, jakiego można się spodziewać:

- rzutowanie na typ całkowity elementu wyliczenia zwraca wartość atrybutu `value` tego elementu;

  ```
  >>> int(Shape.CIRCLE)
  1
  ```

- mogą być składowymi wyrażeń;

  ```
  >>> Shape.CIRCLE + 5
  6
  ```

- mogą być indeksami sekwencji, takich jak np. lista;

  ```
  >>> ["a", "b", "c"][Shape.CIRCLE]
  'b'
  ```

- mogą być argumentami dla funkcji `range`.

  ```
  >>> [j for j in range(Shape.SQUARE)]
  [0, 1]
  ```

Obiekty klasy `Shape` są jednocześnie obiektami klasy `int` oraz klasy `IntEnum`.

```
>>> issubclass(Shape, int)
True
>>> issubclass(Shape, IntEnum)
True
>>> isinstance(Shape.CIRCLE, int)
True
>>> isinstance(Shape.CIRCLE, IntEnum)
True
```

8.3.3. Klasa Flag

Klasa `Flag` jest podklasą klasy `Enum`. Elementy `Flag` można łączyć przy użyciu operatorów bitowych (`&`, `|`, `^`, `~`), ale nie można ich łączyć ani porównywać z żadnym innym

wyliczeniem `Flag` ani `int`. Chociaż możliwe jest bezpośrednie określenie wartości, zaleca się użycie wartości `auto`() i pozwolenie klasie `Flag` na wybranie odpowiedniej wartości.

Jeżeli kombinacja elementów nie powoduje ustawienia żadnej flagi, to wartością logiczną wyniku jest `False`.

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...   RED = auto()
...   BLUE = auto()
...   GREEN = auto()
...
>>> repr( Color.RED & Color.GREEN )
'<Color.0: 0>'
>>> bool( Color.RED & Color.GREEN )
False
```

Poszczególne flagi powinny mieć wartości będące potęgami dwójki (tzn. `1`, `2`, `4` itd.), podczas gdy kombinacje flag nimi nie będą.

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...   RED = auto()                #0001
...   BLUE = auto()               #0010
...   GREEN = auto()              #0100
...   WHITE = RED | BLUE | GREEN  #0111
...
>>> repr( Color.RED )
'<Color.RED: 1>'
>>> repr( Color.BLUE )
'<Color.BLUE: 2>'
>>> repr( Color.GREEN )
'<Color.GREEN: 4>'
>>> repr( Color.WHITE )
'<Color.WHITE: 7>'
```

Utworzenie w wyliczeniu elementu o wartości 0, tzn. „brak ustawionych flag”, oznacza, że jego wartością logiczną jest `False`.

```
>>> from enum import Flag, auto
>>>
```

```
>>> class Color(Flag):
...   BLACK = 0
...   RED = auto()
...   BLUE = auto()
...   GREEN = auto()
...
>>> repr( Color.BLACK )
'<Color.BLACK: 0>'
>>> bool(Color.BLACK)
False
```

8.3.4. Klasa `IntFlag`

Elementy klasy `IntFlag`, która jest podklasą klasy `Enum` i `int`, mogą być łączone za pomocą operatorów bitowych (`&`, `|`, `^`, `~`), a wynikiem jest nadal element `IntFlag`. Ponadto elementy `IntFlag` mogą być używane wszędzie tam, gdzie używane są elementy `int`. Wynik każdej operacji na elementach klasy `IntFlag`, poza operacjami bitowymi, powoduje utratę przynależności tego wyniku do klasy `IntFlag`.

```
>>> from enum import IntFlag
>>>
>>> class Perm(IntFlag):
...   R = 4
...   W = 2
...   X = 1
...
>>> repr(Perm.R | Perm.W)
'<Perm.R|W: 6>'
>>> Perm.R + Perm.W
6
>>> type(Perm.R + Perm.W)
<class 'int'>
>>> RW = Perm.R | Perm.W
>>> type(RW)
<enum 'Perm'>
>>> repr(RW)
'<Perm.R|W: 6>'
>>> Perm.R in RW
True
```

```
>>> Perm.X in RW
False
```

Możliwe jest także nadawanie nazw kombinacjom stałych symbolicznych.

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...   R = 4
...   W = 2
...   X = 1
...   RWX = 7
...
>>> repr( Perm.RWX )
'<Perm.RWX: 7>'
>>> repr( ~Perm.RWX )
'<Perm.-8: -8>'
```

Inną ważną różnicą między `IntFlag` a `Enum` jest to, że jeżeli nie ustawiono żadnych flag (wartość 0), to wartość logiczna wynosi `False`.

```
>>> repr( Perm.R & Perm.X )
'<Perm.0: 0>'
>>> bool(Perm.R & Perm.X)
False
```

Ponieważ instancje klasy `IntFlag` są również instancjami klasy `int`, to można łączyć jedne z drugimi.

```
>>> repr(Perm.X | 8 )
'<Perm.8|X: 9>'
```

Używanie klas `Enum` i `Flag` jest zalecane, ponieważ `IntEnum` i `IntFlag` łamią pewne semantyczne obietnice dla wyliczeń, które mają reprezentować stałe symboliczne, poprzez porównywalność z liczbami całkowitymi, a tym samym przez przechodniość do innych niepowiązanych wyliczeń. Natomiast `IntEnum` i `IntFlag` powinny być używane tylko w przypadkach, gdy `Enum` i `Flag` nie działają, np. gdy stałe całkowite są zastępowane wyliczeniami lub w celu współdziałania z innymi systemami.

8.4. Iteratory

`Iteratorem` nazywamy obiekt pozwalający na sekwencyjny dostęp do wszystkich lub wybranych elementów zawartych w innym obiekcie, zwykle kolekcji lub łańcuchu znaków. Iterator można rozumieć jako rodzaj wskaźnika udostępniającego dwie podstawowe

operacje: odwołanie się do konkretnego elementu w kolekcji (dostęp do elementu) oraz modyfikację samego iteratora tak, by wskazywał na kolejny element (sekwencyjne przeglądanie elementów). Musi także istnieć sposób utworzenia iteratora tak, by wskazywał na pierwszy element oraz sposób określenia, kiedy iterator wyczerpał wszystkie elementy w kolekcji.

W zależności od języka i zamierzonego zastosowania, iteratory mogą dostarczać dodatkowych operacji lub posiadać różne dodatkowe zachowania. Podstawowym celem iteratora jest pozwolić użytkownikowi przetworzyć każdy element w kolekcji bez konieczności zagłębiania się w jej wewnętrzną strukturę. Pozwala to kolekcji przechowywać elementy w dowolny sposób, podczas gdy użytkownik może traktować ją jak zwykłą sekwencję lub listę. Klasa iteratora jest zwykle projektowana wraz z klasą odpowiadającej mu kolekcji i jest z nią ściśle powiązana. Zwykle to kolekcja dostarcza metod tworzących iteratory.

Iteratory są jednym z podstawowych elementów Pythona i często są w ogóle niezauważalne, gdyż są niejawnie wykorzystywane w pętlach `for`. Wszystkie standardowe typy sekwencyjne w Pythonie, jak również wiele klas w bibliotece standardowej, udostępniają iterację. Iteratory można również definiować w sposób jawny. Do pobrania iteratora z kolekcji typu sekwencyjnego wykorzystuje się funkcję wbudowaną `iter()`. Wbudowana funkcja `next()` zwraca przy każdym wywołaniu kolejny element kolekcji oraz modyfikuje iterator, tak aby wskazywał on na następny element kolekcji. Gdy nie ma więcej elementów, funkcja ta zgłasza wyjątek `StopIteration`. Przykład:

```
1 for x in ['a', 'b', 'c']:
2   print(x)
```

Wykonanie powyższej pętli obejmuje:

- `Początek iteracji`: od `obj`, który tu jest listą, zostaje zażądany iterator; `obj` go podaje, umownie oznaczamy go `it`.
- `Pierwszy obrót pętli`:
  1. od `it` żądany jest obiekt;
  2. `it` podaje pierwszy element `obj`, czyli `'a'`;
  3. `'a'` nazywane jest `x`, wykonywana jest treść pętli (`print(x)`).
- `Drugi obrót pętli`: powtarzają się kroki `1-3`, ale tym razem elementem podanym przez `it` jest `'b'`.
- `Trzeci obrót pętli`: jak wyżej, dla elementu `'c'`.
- Pętla, chcąc rozpocząć czwarty obrót, żąda obiektu od `it`.
- `it` nie może już podać obiektu, zgłasza koniec iteracji, pętla się kończy.

Każda iteracja ma swój iterator, co ilustruje poniższy przykład. Dla listy `lst`:

```
>>> lst = ['a', 'b', 'c']
```

Definiujemy iterator `it1`:

```
>>> it1 = iter(lst)
```

A następnie od iteratora `it1` żądamy podania elementu z obiektu `lst` i przesunięcia się na następny element:

```
>>> next(it1)
'a'
```

Definiujemy teraz kolejny iterator `it2`:

```
>>> it2 = iter(lst)
```

Sprawdzamy czy utworzone wcześniej operatory nie są jednym i tym samym:

```
>>> it is it2
False
```

Przekonujemy się, że nie są, więc od każdego z nich żądamy podawania kolejnych elementów sekwencji, a komunikaty na ekranie potwierdzają niezależność iteratorów.

```
>>> next(it2)
'a'
>>> next(it1)
'b'
>>> next(it1)
'c'
>>> next(it2)
'b'
```

Wiedząc, że w chwili, gdy metoda `next` dotrze do końca sekwencji zgłaszany jest wyjątek `StopIteration`, można wykorzystać obsługę wyjątków `try`, aby zapobiec nieoczekiwanemu zakończeniu programu. Dla zilustrowania przebiegu iteracji po elementach łańcucha `sequence`, na listingu 8.6 wykorzystamy w kodzie pętlę `while` i obsługę wyjątków.

LISTING 8.6: *Użycie iteratora i obsługa zgłaszanego wyjątku* `StopIteration`

```
1 sequence = "Ala ma kota!!!"
2 it = iter(sequence)
3 try:
4   while True:
5     val = next(it)
6     print(val)
7 except StopIteration:
8   print("KONIEC")
```

Dowolna zdefiniowana przez użytkownika klasa może udostępniać standardową iterację (niejawną lub jawną), jeśli posiada zdefiniowana w ciele metodę `__iter__()` zwracającą iterator. Zwrócony iterator musi posiadać również zdefiniowane metody `__iter__()` oraz `__next__()`.

W Pythonie obiekty iterowalne (ang. *iterable*) reprezentują te ciągi obiektów, przez które można iterować: przykładami są pętle `for` działające dla obiektów iterowalnych, konstrukcje list lub słowników składanych. Wiele metod lub funkcji z bibliotek standardowych (i nie tylko) jest napisanych tak, aby ich argumentami były dowolne obiekty iterowalne, np. `sorted(iterable)`. W Pythonie, aby obiekt był rozpoznany jako iterowalny, on i jego iteratory muszą realizować tzw. `protokół iteratorów`:

- Obiekt iterowalny `ob` musi implementować metodę specjalną `__iter__()`, zwracającą iterator.
- Iterator musi implementować metody:
  - `__next__()`, która albo zwraca obiekt, albo zgłasza wyjątek reprezentujący koniec iteracji (`StopIteration`);
  - `__iter__()`, zwracającą iterator (może nim być – i zazwyczaj jest – on sam).

Począwszy od Pythona 3.4, najdokładniejszym sposobem sprawdzenia, czy obiekt jest iterowalny, jest wywołanie funkcji `iter()` i obsłużenie wyjątku `TypeError`, jeżeli tak nie jest. Przykłady obiektów iterowalnych:

- łańcuchy znaków (obiekty klasy `str`),
- sekwencje bajtów (obiekty klas `bytes` oraz `bytearray`),
- listy (obiekty klasy `list`),
- zbiory (obiekty klasy `set`),
- krotki (obiekty klasy `tuple`),
- słowniki (obiekty klasy `dict`),
- pliki (obiekty zwracane przez funkcję `open`),
- zakresy (obiekty klasy `range`).

Zauważmy, że obiekty we wszystkich powyższych przykładach są obiektami iterowalnymi, ale nie są iteratorami.

Badanie typów iteratorów wybranych obiektów iterowalnych można przeprowadzić w następujący sposób:

```
for typ in (str, tuple, list, set, dict):
  ob = typ()     # nowy obiekt danego typu
  it = iter(ob) # nowy iterator tego obiektu
  print(type(it))
print(type(iter(range(0)))) # range() niepoprawne
```

```
 OUTPUT:
<class 'str_iterator'>
```

```
<class 'tuple_iterator'>
<class 'list_iterator'>
<class 'set_iterator'>
<class 'dict_keyiterator'>
<class 'range_iterator'>
```

Nazwa iteratora słownika `dict_keyiterator` sugeruje, że iterator służy do iterowania po kluczach słownika. I tak jest w rzeczywistości:

```
>>> d = {'a': 1, 'b': 2, 'c': 42}
>>> for k in d:
...     print(k)
...
a
b
c
```

Słowniki mają metody, które zwracają iterowalne obiekty reprezentujące wartości oraz pary `klucz-wartość`:

```
>>> values = d.values()
>>> print(type(values), type(iter(values)))
<class 'dict_values'> <class 'dict_valueiterator'>
>>> items = d.items()
>>> print(type(items), type(iter(items)))
<class 'dict_items'> <class 'dict_itemiterator'>
>>> # Poniżej iterujemy po d.items(), zwrócone
>>> # obiekty są ,,konsumowane'' przez konstruktor listy
>>> print(list(d.items()))
[('a', 1), ('b', 2), ('c', 42)]
>>>
```

Obiekty będące iteratorami w Pythonie są zgodne z protokołem iteracyjnym, co zasadniczo oznacza, że zapewniają dwie metody: `__iter__()` oraz `__next__()`. Iterator to obiekt reprezentujący strumień danych; ten obiekt zwraca ze strumienia danych jeden element na raz. Iterator Pythona musi obsługiwać metodę o nazwie `__next__()`, która nie przyjmuje argumentów i zawsze zwraca następny element strumienia. Jeżeli nie ma więcej elementów w strumieniu, metoda `__next__()` musi wyrzucić wyjątek `StopIteration`. Iteratory nie muszą być skończone; rozsądnie jest napisać iterator, który generuje nieskończony strumień danych.

Na listingach 8.7 oraz 8.8 przedstawiamy odpowiednio definicję klasy iteratora nieskończonego `PowersOfTwo` generującego kolejne potęgi dwójki oraz program wykorzystujący

ten iterator do wypisywania na ekranie kolejnych potęg liczby 2 (linie 11-14) nie większych od podanego górnego limitu (linia 7 lub 9). Zwracamy uwagę na potrzebę określenia w programie głównym warunku zakończenia działania nieskończonego iteratora.

Listing 8.7: *Klasa iteratora zwracającego kwadraty kolejnych liczb naturalnych*

```
1  # powersoftwo.py
2  class PowersOfTwo:
3    def __init__(self):
4      self.num = 1
5    def __iter__(self):
6      return self
7    def __next__(self):
8      num = self.num
9      self.num *= 2
10     return num
```

Listing 8.8: *Wykorzystanie iteratora klasy* `PowersOfTwo`

```
1  # main_powersoftwo.py
2  from sys import argv
3  from powersoftwo import PowersOfTwo
4
5  def main():
6    try:
7      limit = int(argv[1])
8    except:
9      limit = 10000
10   it = iter(PowersOfTwo())
11   a = next(it)
12   while a < limit:
13     print(a)
14     a = next(it)
15
16 if __name__ == "__main__":
17   main()
```

Jednym z najczęściej wykonywanych działań na listach i innych obiektach iterowalnych jest zastosowanie jakiejś operacji do każdego ich elementu i zebranie wyników. Ponieważ jest to tak często wykonywana operacja, Python udostępnia odpowiednią funkcję wbudowaną, która jest w stanie zrobić to za nas.

- Funkcja `map` służy do zastosowania przekazanej funkcji na każdym elemencie obiektu iterowalnego i zwraca obiekt iteratora zawierający wszystkie wyniki jej wywołania:

```
map(function, *iterables) -> map object
```

Zwracany obiekt iteratora może być przekształcony na listę za pomocą funkcji wbudowanej `list`. Wykorzystamy funkcję `map` do utworzenia listy kwadratów liczb z zakresu od 0 do 5, przy czym dla każdej wartości z tego zakresu zostanie zastosowane wyrażenie lambda obliczające jej kwadrat.

```
>>> list(map(lambda x: x * x, range(6)))
[0, 1, 4, 9, 16, 25]
```

Należy zwrócić szczególną uwagę na liczbę obiektów iterowalnych w wywołaniu funkcji `map`, która musi być równa liczbie argumentów wywoływanej w niej funkcji, np. funkcja standardowa `pow` wymaga podania dwu argumentów: podstawy i wykładnika. Mapowanie powoduje wykonanie następujących działań: `0**0` (w przypadku Pythona nie jest zgłaszany wyjątek dla tego symbolu nieoznaczonego, tylko zostało przyjęte, że wynikiem działania jest 1), `1**1`, ..., `5**5`.

```
>>> potegi = map(pow, range(6), range(6))
>>> list(potegi)
[1, 1, 4, 27, 256, 3125]
```

Działanie funkcji `map` kończy się po wyczerpaniu najkrótszego z obiektów iterowalnych.

```
>>> potegi = map(pow, range(6), range(9))
>>> list(potegi)
[1, 1, 4, 27, 256, 3125]
```

- Funkcja `filter` odfiltrowuje elementy obiektu iterowalnego w oparciu o funkcję testującą:

```
filter(function, iterable) -> filter object
```

Elementy obiektu iterowalnego, dla których funkcja testująca zwraca `True`, dodawane są do listy wyników. Funkcja `filter` zwraca obiekt iteratora zawierający odfiltrowane elementy, który może być przekształcony na listę za pomocą funkcji `list`. Jako przykład przedstawimy wykorzystanie funkcji `filter` do wybrania tylko liczb nieparzystych z różnych obiektów iterowalnych.

```
>>> liczby = [x for x in range(10)]
>>> def odd(x): return x % 2 == 1
>>> nieparzyste = filter(odd, liczby)
>>> print(list(nieparzyste))
[1, 3, 5, 7, 9]
```

```
>>> nieparzyste = filter(odd, range(10))
>>> print(list(nieparzyste))
[1, 3, 5, 7, 9]
>>> nieparzyste = filter(lambda x: x % 2 == 1, range(10))
>>> print(list(nieparzyste))
[1, 3, 5, 7, 9]
```

- Wywołanie funkcji `reduce(function,iterable)` z modułu *functools*, gdzie pierwszy argument `function` jest funkcją dwuargumentową, a drugi `iterable` jest obiektem iterowalnym, zwraca pojedynczą wartość obliczaną następująco:
  - funkcja `function` pobiera dwa pierwsze elementy obiektu `iterable` i oblicza wynik;
  - następnie funkcja `function` pobiera poprzedni wynik oraz trzeci element obiektu `iterable` i oblicza wynik;

    ...
  - funkcja `function` pobiera poprzedni wynik oraz ostatni element z obiektu `iterable` i oblicza wynik.

  Działanie funkcji `reduce` można zasymulować definiując następującą funkcję:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Pokażemy kilka przykładów użycia funkcji `reduce` dla różnych obiektów iterowalnych wykonując na nich podstawowe operacje arytmetyczne oraz dla łańcuchów konkatenację ich znaków wraz z odwróceniem ich kolejności.

```
>>> from functools import reduce
>>> reduce(lambda  x, y: x + y, [1, 2, 3, 4])
10
>>> reduce(lambda  x, y: x * y, [1, 2, 3, 4])
24
>>> reduce(lambda  x, y: x * y, {1, 2, 3, 4})
24
>>> reduce(lambda  x, y: x + y, range(1,5))
10
```

```
>>> s = "Kiler skazany na dobre zmiany"
>>> reduce(lambda x, y: y + x, s)
'ynaimz erbod an ynazaks reliK'
>>> reduce(lambda x, y: x + y, s)
'Kiler skazany na dobre zmiany'
```

8.4.1. Moduł `itertools`

Moduł `itertools` (dokumentacja modułu dostępna jest pod linkiem: `https://docs.python.org/3/library/itertools.html`) implementuje szereg iteratorów zainspirowanych konstrukcjami z języków `APL`, `Haskell` i `SML`. Moduł standaryzuje podstawowy zestaw szybkich, wydajnych pamięciowo narzędzi, które są przydatne samodzielnie lub w połączeniu. Razem tworzą one „`algebrę iteratorów`", umożliwiającą konstruowanie wyspecjalizowanych narzędzi w sposób zwięzły i wydajny w czystym Pythonie.

Przykładowo, język `SML` udostępnia narzędzie `tabulate(f)`, które tworzy sekwencję `f(0), f(1), ...`. Ten sam efekt można osiągnąć w Pythonie, łącząc funkcje `map()` oraz `count()`, tworząc `map(f, count())`.

Narzędzia modułu `itertools` oraz ich wbudowane odpowiedniki dobrze współpracują również z szybkimi funkcjami w module `operator`. Przykładowo, operator mnożenia może zostać odwzorowany na dwa wektory, tworząc wydajny iloczyn skalarny: `sum(map(operator.mul, vector1, vector2))`.

```
>>> import operator
>>> a = [1,2,3]
>>> b = [3,4,5]
>>> sum(map(operator.mul, a, b))
26
```

Opisy funkcji z modułu `itertools`, które poniżej przedstawiamy, wzbogaciliśmy przykładami ich działania wykonanymi w interpreterze. Aby prawidłowo wykonać każdy z przykładów, należy po uruchomieniu interpretera zaimportować niezbędne moduły:

```
>>> from itertools import *
>> import operator
```

- Funkcja `count(start=0, step=1)` tworzy iterator, który zwraca wartości, począwszy od tej podanej jako parametr `start`. Kolejne są zwiększane przez wartość `step`, która jest drugim parametrem.

  ```
  count(10) --> 10 11 12 13 14 ...
  count(2.5, 0.5) --> 2.5 3.0 3.5 ...
  count(2, -3) --> 2 -1 -4 -7 -10 -13 ...
  ```

Wprowadzamy teraz ograniczenie dolne dla zwracanych przez `count` wartości, przekroczenie którego powoduje przerwanie działania pętli.

```
>>> for num in count(2,-3):
...     print(num, end=", ")
...     if num < -20:
...             break
...
2, -1, -4, -7, -10, -13, -16, -19, -22,
```

- Funkcja `cycle(iterable)` przyjmuje jako parametr obiekt `iterable`, przez który iterator ten przechodzi. Przy czym, jeśli obiekt się wyczerpie (dojdzie do końca) to nadal będzie zwracał wartości od początku. Dzieje się tak, bo `cycle` zapisuje sobie każdy element w pamięci.

```
>>> c = cycle("ABCD")
>>> for i in range(10):
...     print(next(c), end=", ")
...
A, B, C, D, A, B, C, D, A, B,
```

- Funkcja `repeat(object, times=None)` zwraca dany obiekt `times` razy. Jeśli parametr `times` nie jest przekazany, to obiekt będzie zwracany nieskończenie długo. Oficjalna dokumentacja tłumaczy, że `repeat` używa się głównie jako stały argument przekazywany do funkcji `map`. Może też służyć do dodawania stałej wartości do krotki.

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Funkcja `accumulate(iterable, func = operator.add)` tworzy iterator, który zwraca skumulowane sumy lub skumulowane wyniki innych funkcji binarnych, określone za pomocą opcjonalnego argumentu `func`. Elementy obiektu iterowalnego `iterable` mogą być dowolnego typu, który dopuszcza funkcja `func`. Przykładowo, dla domyślnej operacji dodawania elementy mogą być dowolnymi typami. Jeżeli argument `iterable` nie posiada elementów, to wynik też nie będzie posiadać elementów.

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
```

- Funkcja `chain(*iterables)` zwraca obiekt klasy `chain`, którego metoda `__next__` zwraca elementy z pierwszego obiektu iterowalnego, aż do jego wyczerpania, a następnie elementy z następnego obiektu iterowalnego, aż do wyczerpania wszystkich obiektów.
  `chain.from_iterable(iterable)` - alternatywny konstruktor klasy `chain` przyjmujący pojedynczy argument iterowalny.

```
>>> list(chain("Ala", "ma", "kota"))
['A', 'l', 'a', 'm', 'a', 'k', 'o', 't', 'a']
>>> list(chain.from_iterable(["Ala", "ma", "kota"]))
['A', 'l', 'a', 'm', 'a', 'k', 'o', 't', 'a']
```

- Funkcja `compress(iterable, selector)` tworzy iterator, który filtruje elementy z obiektu iterowalnego `iterable`, zwracając tylko te, które mają odpowiadający im element w selektorze `selector` wartościowany jako `True`. Zatrzymuje się po wyczerpaniu obiektu `iterable` lub obiektu `selector`.

```
>>> list(compress("ABCDEF", [1, 0, 1, 0, 1, 1]))
['A', 'C', 'E', 'F']
>>> list(compress(count(2, 3), repeat(True, 8)))
[2, 5, 8, 11, 14, 17, 20, 23]
>>> list(compress(cycle("ABC"), repeat(True, 8)))
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B']
```

- Funkcja `dropwhile(predicate, iterable)` tworzy iterator, który usuwa elementy z obiektu iterowalnego tak długo, jak długo predykat jest prawdziwy, a następnie zwraca każdy element. Iterator ten nie wytwarza żadnego wyjścia, dopóki predykat nie stanie się fałszywy, więc jego czas uruchamiania może być długi.

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8]
>>> list(dropwhile(lambda x: x < 8, L))
[8, 1, 2, 3, 4, 5, 6, 7, 8]
>>> L = [x for x in range(50000001)]
>>> list(dropwhile(lambda x: x < 50000000, L))
[50000000]
```

- Funkcja `filterfalse(predicate, iterable)` tworzy iterator filtrujący elementy z obiektu `iterable`, zwracający tylko te, dla których `predicate` jest fałszywy. Jeżeli predykat jest równy `None`, zwraca elementy, które są fałszywe.

```
>>> list(filterfalse(lambda x: True, range(10)))
[]
```

```
>>> list(filterfalse(lambda x: False, range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(filterfalse(lambda x: None, range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(filterfalse(lambda x: x % 2, range(10)))
[0, 2, 4, 6, 8]
>>> list(filterfalse(bool, range(10)))
[0]
>>> list(filterfalse(None, range(10)))
[0]
```

- Funkcja `groupby(iterable, key=None)` tworzy iterator, który zwraca kolejne klucze i grupy z `iterable`. Argument `key` jest funkcją, która oblicza wartość dla każdego elementu. Jeżeli argument `key` nie jest określony lub ma wartość `None`, to argument ten staje się domyślnie funkcją identycznościową. Zasadniczo, argument `iterable` powinien być już posortowany przez funkcję `key`.

```
>>> L = [k for k, g in groupby('AAAABBBCCDAABBB')]
>>> L
['A', 'B', 'C', 'D', 'A', 'B']
>>> L = [list(g) for k, g in groupby('AAAABBBCCDAABBB')]
>>> L
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D'],
['A', 'A'], ['B', 'B', 'B']]
```

- Funkcje:
  `islice(iterable, start, stop[, step])`
  `islice(iterable, stop)`
  tworzą iteratory służące do pobierania wycinków danych z obiektu iterowalnego.

```
>> "".join(islice('ABCDEFG', 2))
'AB'
>>> "".join(islice('ABCDEFG', 2, 4))
'CD'
>>> "".join(islice('ABCDEFG', 2, None))
'CDEFG'
>>> "".join(islice('ABCDEFG', 0, None, 2))
'ACEG'
>>> list(islice(count(0), 10, 30, 2))
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

- Funkcja `starmap(function, iterable)` tworzy iterator, zwracający wyniki zastosowania funkcji `function` do argumentów uzyskanych z obiektu iterowalnego, którego elementami są krotki o długości równej liczbie argumentów funkcji.

```
>>> list(starmap(operator.add,[(2,3),(4,5),(1,1)]))
[5, 9, 2]
```

- Funkcja `takewhile(predicate, iterable)` tworzy iterator, który zwraca te elementy z iteracji, dla których predykat jest prawdziwy.

```
>>> list(takewhile(lambda x: x < 5,[1,4,6,4,1]))
[1, 4]
>>> list(takewhile(lambda x: x < 9, count(3)))
[3, 4, 5, 6, 7, 8]
```

- Funkcja `tee(iterable, n = 2)` zwraca w postaci krotki `n` niezależnych iteratorów z pojedynczego obiektu iterowalnego.

```
>>> t = tee('abcdefgh', 3)
>>> list(t[0])
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> list(t[1])
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> list(t[2])
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

- Funkcja `zip_longest(*iterables, fillvalue=None)` tworzy iterator, który agreguje elementy z każdego obiektu iterowalnego. Jeżeli liczba iteracji jest nierównomiernej długości, brakujące wartości są wypełniane wartością wypełnienia. Iteracja trwa do momentu wyczerpania najdłuższej iteracji.

```
>>> list(zip_longest("ABCD", "xy", fillvalue="-"))
[('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
```

- Funkcja `product(*iterables, repeat=1)` tworzy iloczyn kartezjański obiektów iterowalnych.

```
>>> for elem in product("ABCD", "xy"):
...     print(elem)
...
('A', 'x')
('A', 'y')
('B', 'x')
```

```
('B', 'y')
('C', 'x')
('C', 'y')
('D', 'x')
('D', 'y')
>>> for elem in product("ABCD", "xy", repeat = 2):
...     print(elem)
...
('A', 'x', 'A', 'x')
('A', 'x', 'A', 'y')
('A', 'x', 'B', 'x')
('A', 'x', 'B', 'y')
('A', 'x', 'C', 'x')
('A', 'x', 'C', 'y')
('A', 'x', 'D', 'x')
('A', 'x', 'D', 'y')
('A', 'y', 'A', 'x')
('A', 'y', 'A', 'y')
('A', 'y', 'B', 'x')
('A', 'y', 'B', 'y')
('A', 'y', 'C', 'x')
('A', 'y', 'C', 'y')
('A', 'y', 'D', 'x')
('A', 'y', 'D', 'y')
... # itd.
>>> for elem in product(range(2), repeat=3):
...     print(elem)
...
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
```

- Funkcja `permutations(iterable, r=None)` tworzy krotki o długości `r` wszystkich możliwych uporządkowań, bez powtarzających się elementów.

```
>>> for elem in permutations("ABC", 2):
...     print(elem)
...
('A', 'B')
('A', 'C')
('B', 'A')
('B', 'C')
('C', 'A')
('C', 'B')
>>> for elem in permutations(range(2)):
...     print(elem)
...
(0, 1)
(1, 0)
```

- Funkcja `combinations(iterable, r)` tworzy podzbiory (pokazywane jako krotki), czyli kombinacje o długości `r` z elementów obiektu `iterable`.

```
>>> for elem in combinations("ABCD", 2):
...     print(elem)
...
('A', 'B')
('A', 'C')
('A', 'D')
('B', 'C')
('B', 'D')
('C', 'D')
>>> list(combinations(range(4), 3))
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
```

- Funkcja `combinations_with_replacement(iterable, r)` tworzy podciągi o długości `r` z elementów obiektu `iterable`, pozwalając na powtarzanie poszczególnych elementów (wariacje).

```
>>> for elem in combinations_with_replacement("ABC", 2):
...     print(elem)
...
('A', 'A')
('A', 'B')
('A', 'C')
('B', 'B')
```

```
('B', 'C')
('C', 'C')
```

8.5. Generatory

Generatory są szczególnym przypadkiem iteratorów, które dzięki koncepcji iteracji pozwalają na pisanie programów o lepszej wydajności. Wyróżniamy dwa typy generatorów:

- `Funkcje generatorowe`: podobne są do zwyczajnych funkcji, przy czym zamiast zwracać wynik poprzez instrukcję `return` używają specjalnej instrukcji `yield`, która pozwala im zawiesić i wznowić swój stan pomiędzy kolejnymi wywołaniami.
- `Wyrażenia generatorowe`: podobne są do list składanych, przy czym zamiast zwracać listę, zwracają obiekt, który tworzy wyniki po kolei. Składnia jest taka sama jak dla list składanych, tylko zamiast nawiasów kwadratowych używa się nawiasów okrągłych.

Słowa kluczowego `yield` używa się podobnie do `return`. Wykonanie instrukcji `return` przekazuje na stałe kontrolę do miejsca jej wywołania. Z kolei wykonanie instrukcji `yield` przekazuje kontrolę, ale tylko tymczasowo, gdyż „usypia" funkcję, w której została wywołana i zapamiętuje jej stan lokalny. Każda funkcja zawierająca instrukcję `yield` jest funkcją generatora, a każdy generator jest iteratorem.

Kiedy funkcja generatora jest wywoływana, argumenty są przypisywane do lokalnego kontekstu funkcji (jak w normalnych funkcjach), ale kod wewnątrz funkcji nie zostaje wykonany. Funkcja generatorowa zwraca obiekt generatora będący iteratorem. Za każdym wywołaniem na tym obiekcie funkcji `next()`, kod funkcji generatora jest wykonywany do napotkania instrukcji `yield` albo `return` albo do końca funkcji. Gdy napotkaną instrukcją jest `yield`, to stan funkcji generatora jest zamrażany, a wartość po prawej stronie `yield` jest zwracana do kodu wykonującego funkcję `next()`. Jeżeli natomiast generator wyjdzie w inny sposób niż przy użyciu instrukcji `yield`, to już na stałe przestaje generować nowe wartości. Nie trzeba więc zgłaszać wyjątku `StopIteration`, co jest na pewno istotnym udogodnieniem. Ponieważ obiekt zwracany przez funkcję generatorową jest iteratorem, w prosty sposób można użyć takiej funkcji, np. w pętli `for`, co pokazano na listingu 8.9. Funkcja generatorowa `gen_squares` generuje po kolei kolejne kwadraty liczb całkowitych nieujemnych.

LISTING 8.9: *Przykład prostej funkcji generatorowej*

```
1 def main():
2   gen = gen_squares(6)
3   print(type(gen))
4   for num in gen:
5     print(num, end=" ")
```

```
6 def gen_squares(n):
7   for j in range(n):
8     yield j * j
9 if __name__ == '__main__':
10  main()
```

Gdybyśmy nie zastosowali funkcji generatorowej, musielibyśmy zbudować w funkcji od razu listę wszystkich uzyskanych wartości, co w przypadku dużej liczby wyników powoduje dużą objętość pamięci i zajmuje dużo czasu. Generatory pozwalają nie tylko zaoszczędzić pamięć, ale również równomiernie rozłożyć czas generowania danych, pozwalając w tym czasie innemu kodowi przetwarzać już wygenerowane dane cząstkowe. W przypadku bardziej zaawansowanych zastosowań generatory mogą stanowić alternatywę dla ręcznego zapisywania stanu między iteracjami w obiektach klasy. W przypadku generatorów zmienne dostępne w zasięgach funkcji są automatycznie zapisywane i przywracane.

LISTING 8.10: *Przykład użycia funkcji generatorowej jako iteratora nieskończonego*

```
1 def gen_squares(start):
2   while True:
3     yield start * start
4     start += 1
5 g = gen_squares(10)
6 print(type(g))
7 for j in gen_squares(10):
8   print(j, end=" ")
9   if j > 300:
10    break
```

Generatory można tworzyć również przez krótkie **wyrażenia generatorowe**, które mają postać podobną do list składanych, przy czym zamiast tworzyć listę, tworzą generator „leniwie” zwracający kolejne obiekty. Na przykład listę pierwszych ośmiu kwadratów możemy skonstruować jako listę składaną:

```
>>> a = [num * num for num in range(8)]
>>> a
[0, 1, 4, 9, 16, 25, 36, 49]
```

Można także zdefiniować wyrażenie generatorowe, które nie przechowuje wszystkich wartości, tylko będąc iteratorem dzięki funkcji **next** w danym momencie zwraca jedną wartość.

```
>>> g = (num * num for num in range(8))
>>> g
<generator object <genexpr> at 0x7f47bcb8b220>
```

```
>>> next(g)
0
```

Wyrażenia generatorowe są alternatywą dla innych obiektów iterowalnych lub kontenerów, które wymagają więcej miejsca w pamięci, takich jak listy, krotki lub zbiory. Do zilustrowania różnicy wykorzystamy funkcje `getsizeof` z modułu `sys` zwracającą rozmiar obiektu w bajtach oraz funkcję `randrange` z modułu `random`, które należy zaimportować przed przystąpieniem do wykonania poniższych instrukcji. Dla listy składanej otrzymujemy `89_095_160 B`:

```
>>> a = [randrange(1,1000) for j in range(10000000)]
>>> sys.getsizeof(a)
89095160
```

Natomiast dla wyrażenia generatorowego jest to tylko `104 B`

```
>>> g = (randrange(1,1000) for j in range(10000000))
>>> sys.getsizeof(g)
104
```

Użycie generatora zmniejsza `856_684` razy zajętość pamięci.

```
>>> sys.getsizeof(a)//sys.getsizeof(g)
856684
```

W Pythonie 3.3 wprowadzono rozszerzoną składnię instrukcji `yield`, która za pomocą klauzuli `from generator` pozwala na delegowanie działania do subgeneratora. W prostych przypadkach jest to odpowiednik pętli `for`, zastosowanie której pokazujemy na listingu 8.11, gdzie wywołanie funkcji `list` zmusza generator do wygenerowania od razu wszystkich wartości.

LISTING 8.11: *Przykład użycia pętli* `for` *w generatorze*

```
1  def main():
2    print(list(both(6)))
3
4  def both(n):
5    for j in range(n): yield j
6    for j in (x ** 2 for x in range(n)): yield j
7
8  if __name__ == '__main__':
9    main()
10
11 # OUTPUT:
12 # [0, 1, 2, 3, 4, 5, 0, 1, 4, 9, 16, 25]
```

Użycie klauzuli `from generator` sprawia, że kod staje się bardziej zwięzły i czytelny oraz obsługuje wszystkie zwykłe konteksty użycia generatora, co zilustrowaliśmy na listingu 8.12.

LISTING 8.12: *Przykład użycia klauzuli* `from` *w generatorze*

```
def main():
  print(list(both(6)))
  print(" : ".join(str(j) for j in both(6)))

def both(n):
  yield from range(n)
  yield from (x ** 2 for  x in range(n))

if __name__ == '__main__':
  main()

# OUTPUT:
# [0, 1, 2, 3, 4, 5, 0, 1, 4, 9, 16, 25]
# 0 : 1 : 2 : 3 : 4 : 5 : 0 : 1 : 4 : 9 : 16 : 25
```

Rozważmy przykłady wyrażenia generatorowego `generator_1` oraz funkcji generatorowej `generator_2`.

```
>>> generator_1 = (x for x in [1, 2, 3, 4, 5])
>>> def generator_2():
...   yield 1
...   yield 2
...   yield 3
...   yield 4
...   yield 5
...
```

Skorzystamy z modułu `collections.abc` i funkcji `Iterator`, `Iterable` oraz `Generator`, aby pokazać wzajemne zależności typów obiektów.

```
>>> from collections.abc import Iterator, Iterable, Generator
>>> type(generator_1)
<class 'generator'>
>>> type(generator_2)
<class 'function'>
>>> type(generator_2())
<class 'generator'>
```

```
>>> isinstance(generator_1, Generator)
True
>>> isinstance(generator_2(), Generator)
True
>>> isinstance(generator_1, Iterator)
True
>>> isinstance(generator_2(), Iterator)
True
>>> isinstance(generator_1, Iterable)
True
>>> isinstance(generator_2(), Iterable)
True
>>> issubclass(Generator, Iterator)
True
```

Ciekawą właściwością generatora jest możliwość wstrzyknięcia mu wartości. Instrukcja `yield` nie tylko zwraca wartość, ale również może przyjmować wartość z zewnątrz. Aby wysłać „coś" do generatora wystarczy wykonać metodę `send()`.

Na listingu 8.13 przedstawiamy przykład generatora, który co `0.5` sekundy generuje kolejną liczbę całkowitą, zaczynając od jedynki. W pętli zdefiniowany zostanie warunek, że jeśli wygenerowana zostanie liczba 10, to do generatora zostanie wysłana liczba `-1` (linia 8). Jest to interpretowane przez generator jako skok używany do generowania kolejnej liczby, co sprawi, że generator policzy od 1 do 10, a następnie zawróci, by zakończyć się na jedynce. W celu zakończenia generatora zostanie użyte słówko `return`.

Listing 8.13: *Przykład użycia metody* `send`

```
1  from time import sleep
2
3  def main():
4    generator = generator_4()
5    for x in generator:
6      print(x)
7      if x == 10:
8        print(generator.send(-1))
9
10 def generator_4():
11   i = 1
12   step = 1
13   while True:
14     new_step = yield i
```

```
15      if new_step:
16        step = new_step
17      i += step
18      if i == 0:
19        return
20      sleep(0.5)
21
22  if __name__ == '__main__':
23    main()
```

Generatory to dużo prostszy sposób na tworzenie iteratorów, szczególnie zamiast tych zbudowanych na podstawie klasy. W znacznej większości przypadków jest to możliwe. Najważniejszą zaletą płynącą z iteratorów i generatorów jest oszczędność zasobów. Zaleca się, aby do funkcji, które oczekują obiektów iterowalnych, takich między innymi jak `min()`, `max()` i `sum()`, przekazywać wyrażenia generatorowe zamiast list składanych. Jest to nie tylko bardziej wydajne, ale również mieści się w „filozofii" Pythona.

8.6. Zadania do samodzielnego rozwiązania

8.6.1. Iteratory

1. Napisz program, który dla danej listy miast wypisze te miasta stosując iterator oraz instrukcję iteracyjną `while`. Wykorzystaj wyjątek `StopIteration`.
2. Napisz program `biglist.py`, który dla liczby `n` tworzy listę liczb od `1` do `n`, a następnie oblicza sumę liczb z tej listy. Program wywołaj w następujący sposób:
   `/usr/bin/time -f "%e sec. %M kB" python biglist.py n`
   gdzie n jest konkretną liczbą.
3. Napisz program `bigrange.py`, który dla liczby `n` oblicza sumę liczb od `1` do `n`. Nie twórz listy, lecz wykorzystaj wbudowaną funkcję `range`. Program wywołaj w następujący sposób:
   `/usr/bin/time -f "%e sec. %M kB" python bigrange.py n`
   gdzie `n` jest konkretną liczbą.
4. Korzystając z informacji dostarczonej przez program `/usr/bin/time` dla dwóch powyższych programów, wyciągnij odpowiednie wnioski.
5. Napisz program, który dla danej listy temperatur w skali Celsjusza, przekształci tę listę w listę temperatur w skali Fahrenheita. Użyj wyrażenia `lambda` oraz wbudowanej funkcji `map`.
6. Napisz program, który dla danej listy temperatur w skali Fahrenheita, przekształci tę listę w listę temperatur w skali Celsjusza. Użyj wyrażenia `lambda` oraz wbudowanej funkcji `map`.

7. Napisz program, który wygeneruje listę liczb Fibonacciego o podanej długości, a następnie stosując wyrażenie `lambda` oraz wbudowaną funkcję `filter` utworzy listę nieparzystych liczb Fibonacciego.
8. Napisz program, który wygeneruje listę liczb Fibonacciego o podanej długości, a następnie stosując wyrażenie lambda oraz wbudowaną funkcję `filter` utworzy listę parzystych liczb Fibonacciego.
9. Napisz program, który dla danej listy liczb znajdzie największą liczbę stosując wbudowaną funkcję `max` oraz funkcję `reduce` z modułu `functools`.
10. Napisz program, który dla danej listy liczb znajdzie największą liczbę stosując odpowiednie wyrażenie lambda oraz funkcję `reduce` z modułu `functools`. W wywołaniu funkcji `reduce` nie korzystaj z wbudowanej funkcji `max`.
11. Napisz program, który dla listy składanej stu losowych liczb z przedziału `<-10,10>` (`lista1`), przekształci tę listę w listę `lista2` kwadratów tych liczb. Użyj wyrażenia `lambda` oraz wbudowanej funkcji `map`. Następnie oblicz i wypisz na ekranie sumę elementów listy `lista2` stosując `iterator` oraz instrukcję iteracyjną `while` i wykorzystując wyjątek `StopIteration`.

8.6.2. Moduł itertools

1. Zdefiniuj następujące funkcje korzystając z funkcji z modułu `itertools` i umieść je w pliku o nazwie `iterutils.py`:
   (a) `pobierz(n, iterable)` - zwraca listę złożoną z n początkowych elementów obiektu iterowalnego `iterable`. Gdy `n > len(iterable)` zwraca listę złożoną ze wszystkich elementów `iterable`.
   (b) `dolacz(value, iterator)` - zwraca iterator, którego pierwszą zwracaną wartością jest `value`, a następnymi są wartości zwracane przez iterator `iterator`.
   (c) `tabela(func, start=0)` - zwraca iterator, którego kolejne wartości to: `func(start)`, `func(start + 1)` itd.
   (d) `nty(iterable, n, default=None)` - zwraca `n`-ty element z obiektu iterowalnego `iterable` lub wartość `default`.
   (e) `takie_same(iterable)` - zwraca wartość `True`, jeżeli wszystkie elementy są sobie równe.
   (f) `policz(iterable, pred=bool)` - zwraca liczbę elementów obiektu iterowalnego `iterable`, dla których predykat `pred` jest prawdziwy.
   (g) `wypelnij(iterable)` - zwraca iterator dostarczający kolejne elementy obiektu iterowalnego `iterable`, a następnie wartość `None` w nieskończoność.
   (h) `nrazy(iterable, n)` - zwraca iterator, powtarzający elementy obiektu iterowalnego `n` razy.
   `list(nrazy("ab", 3) -> ["a", "b", "a", "b", "a", "b"]`

(i) `splaszcz(list_of_lists)` - zwraca iterator spłaszczający jeden poziom zagnieżdżania.

```
list(splaszcz([[1, 2, 3], [5, 6]])) -> [1, 2, 3, 5, 6]
```

(j) `powtarzaj(func, times=None, *args)` - zwraca iterator powtarzający wywołania funkcji `func` z określonymi argumentami.

```
powtarzaj(random.random)
list(powtarzaj(math.sin, 3, math.pi/2)) -> [1.0, 1.0, 1.0]
list(powtarzaj(math.pow, 4, 2, 3)) -> [8.0, 8.0, 8.0, 8.0]
```

2. Napisz program `test_iterutils.py`, a w nim, w funkcji `main`, przetestuj funkcje z pliku `iterutils.py`.

8.6.3. Funkcje generatorowe

1. Napisz program z funkcją generatorową `podzielne_przez_3_i_5(limit)`. Funkcja ta powinna dostarczać kolejne liczby naturalne z przedziału `[0, limit)`, które są podzielne przez `3` i przez `5`. W funkcji `main` wczytaj liczbę naturalną do zmiennej n i wypisz wszystkie liczby zwracane przez funkcję `podzielne_przez_3_i_5` wywołaną z argumentem n.
2. Napisz program z bezargumentową funkcją generatorową, która powinna dostarczać kolejne liczby naturalne podzielne przez `3` i przez `5` (`podzielne_przez_3_i_5`). W funkcji `main` wczytaj liczbę naturalną do zmiennej `n` i wypisz wszystkie liczby dostarczane przez funkcję `podzielne_przez_3_i_5`, które są mniejsze od `n`.
3. Napisz program z funkcją generatorową `fibonacci(limit)`. Funkcja ta powinna dostarczać kolejne liczby Fibonacciego z przedziału `[0,limit)`. W funkcji `main` wczytaj liczbę naturalną do zmiennej `n` i wypisz wszystkie liczby dostarczane przez funkcję `fibonacci` wywołaną z argumentem `n`.
4. Napisz program z funkcją generatorową `fibonacci`. Funkcja ta powinna dostarczać kolejne liczby Fibonacciego. W funkcji `main` wczytaj liczbę naturalną do zmiennej `n` i wypisz wszystkie liczby dostarczane przez funkcję `fibonacci`, które są mniejsze od `n`.

8.6.4. Wyrażenia generatorowe

1. Napisz program, który stosując wyrażenie generatorowe utworzy listę liczb całkowitych z przedziału `[1..n]` podzielnych przez `3` lub podzielnych przez `5`, gdzie `n` jest liczbą podaną jako pierwszy argument programu.
2. Napisz program, który stosując wyrażenie generatorowe utworzy `n`-elementową listę `liczby` liczb typu `float` z przedziału `[-100..100]` zaokrąglonych do dwóch miejsc po przecinku (użyj funkcji `random.uniform`). Następnie wypisz na ekran li-

stę `liczby` oraz stosując wyrażenia generatorowe do każdego z poniższych punktów napisz jedną instrukcję wypisującą:

a) listę dodatnich liczb z listy `liczby`;
b) listę dodatnich liczb z listy `liczby` przekształconych do typu `int`;
c) listę wartości funkcji `math.floor` dla liczb z listy `liczby`;
d) listę wartości funkcji `math.ceil` dla liczb z listy `liczby`;
e) listę wartości funkcji `math.log` dla dodatnich liczb z listy `liczby`.

8.6.5. Zadania dodatkowe

1. Napisz program, który:
   a) korzystając z funkcji (`generuj(n)`), generuje ciąg liczb od `1` do `n`;
   b) korzystając z wyrażenia generującego tworzy ciąg liczb od `1` do `n`.

   Następnie w funkcji `main` wypisz co drugą wygenerowaną liczbę dla każdego z podpunktów osobno:
   1) stosując `iterator` oraz instrukcję iteracyjną `while` i wykorzystując wyjątek `StopIteration`;
   2) stosując pętlę `for`.
2. Napisz program, który z wykorzystaniem nieskończonego generatora o nagłówku: `wielokrotnosc(n)`, znajduje kolejne wielokrotności liczby `n` przekazanej jako argument. Następnie wypisz 10 takich liczb oddzielonych przecinkami:
   a) stosując `iterator` oraz instrukcję iteracyjną `while` i wykorzystując wyjątek `StopIteration`;
   b) stosując pętlę `for`.

Literatura

[1] Beazley David, Brian K. Jones, *Python. Receptury*, Wydanie III, Helion, Gliwice, 2014.

[2] Dawson Michael, *Python dla każdego. Podstawy programowania*, Wydanie III, Helion, Gliwice, 2014.

[3] Downey Allen B., *Myśl w języku Python! Nauka programowania*, Wydanie II, Helion, Gliwice, 2017.

[4] Lutz Mark, *Python. Leksykon kieszonkowy*, Wydanie IV, Helion, Gliwice, 2012.

[5] Lutz Mark, *Python. Wprowadzenie*, Wydanie V, O'Reilly Media, Inc, USA, 2013.

[6] Matthes Eric, *Python. Instrukcje dla programisty dla każdego. Podstawy programowania*, Wydanie II, Helion, Gliwice, 2020.

[7] Ramalho Luciano, *Zaawansowany Python, Przejrzyste, zwięzłe i efektywne programowanie*, Wydanie II, Helion, Gliwice, 2022.

[8] Slatkin Brett, *Efektywny Python. 59 sposobów na lepszy kod*, Helion, Gliwice, 2015.

[9] Summerfield Mark, *Python 3. Kompletne wprowadzenie do programowania*, Wydanie II, Helion, Gliwice, 2010.

[10] Wentworth Peter, Elkner Jeffrey, Downey Allen B., Meyers Chris, *How to Think Like a Computer Scientist: Learning with Python 3*, `http://openbookproject.net/thinkcs/python/english3e/`

[11] `https://www.python.org`

[12] `https://www.python.org/doc/`

[13] `https://docs.python.org/pl/3/tutorial/index.html`

Wydawnictwo Naukowe Uniwersytetu Jana Długosza w Częstochowie
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl