PODSTAWY PROGRAMOWANIA W JĘZYKU PYTHON

W PRZYKŁADACH Z ROZWIĄZANIAMI

ANNA ŁUPIŃSKA-DUBICKA
ANDRZEJ CHMIELEWSKI

Politechnika Białostocka

PODSTAWY PROGRAMOWANIA W JĘZYKU PYTHON
W PRZYKŁADACH Z ROZWIĄZANIAMI

Anna Łupińska-Dubicka
Andrzej Chmielewski

OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2023

Recenzent:
dr inż. Mariusz Rybnik

Redaktor naukowy dyscypliny informatyka techniczna i telekomunikacja:
prof. dr hab. Jarosław Stepaniuk

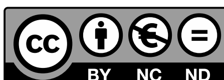
Korekta językowa:
Edyta Chrzanowska

Skład i opracowanie graficzne:
Anna Lupińska-Dubicka

Okładka: Marcin Dominów
Zdjęcie na okładce: wilhei
https://pixabay.com/pl/photos/puzzle-pasuje-zgodzi%c4%87-si%c4%99-brakuje-693870/

© Copyright by Politechnika Białostocka, Białystok 2023

ISBN 978-83-67185-66-0 (eBook)
DOI: 10.24427/978-83-67185-66-0



Publikacja jest udostępniona na licencji
Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 4.0
(CC BY-NC-ND 4.0).
Pełną treść licencji udostępniono na stronie
creativecommons.org/licenses/by-nc-nd/4.0/legalcode.pl.
Publikacja jest dostępna w Internecie na stronie Oficyny Wydawniczej PB.

Oficyna Wydawnicza Politechniki Białostockiej
ul. Wiejska 45C, 15-351 Białystok
e-mail: oficyna.wydawnicza@pb.edu.pl
www.pb.edu.pl

Spis treści

1 Instrukcje wejścia–wyjścia 7

1.1 Zadanie 1 – obliczanie średniej dwóch liczb 8
1.2 Zadanie 2 – wyznaczanie pola kwadratu 9
1.3 Zadanie 3 – symulacja rzutu kostką 10

2 Instrukcje warunkowe 11

2.1 Zadanie 1 – sprawdzanie parzystości liczby 11
2.2 Zadanie 2 – sprawdzanie, czy rok jest przestępny 12
2.3 Zadanie 3 – prosta w układzie współrzędnych 13
2.4 Zadanie 4 – wystawianie oceny 14

3 Instrukcje iteracyjne 17

3.1 Zadanie 1 – zliczanie parzystych 22
3.2 Zadanie 2 – suma poprzedników 23
3.3 Zadanie 3 – suma liczb nieparzystych z przedziału 24
3.4 Zadanie 4 – suma mniejsza od 100 26
3.5 Zadanie 5 – podaj hasło 27
3.6 Zadanie 6 – zgadnij liczbę 28
3.7 Zadanie 7 – losowe liczby podzielne przez 3 30
3.8 Zadanie 8 – liczba pierwsza 31
3.9 Zadanie 9 – obliczanie sumy cyfr liczby 33

4 Typ napisowy 35

4.1 Zadanie 1 – wyznaczanie długości napisu 35
4.2 Zadanie 2 – zliczanie wystąpień znaków 36
4.3 Zadanie 3 – wyświetlanie podłańcuchów 37
4.4 Zadanie 4 – sprawdzanie, czy napis jest palindromem 38
4.5 Zadanie 5 – obliczanie sumy cyfr liczby 40
4.6 Zadanie 6 – generator haseł 41
4.7 Zadanie 7 – anonimizacja danych 42

5 Listy 44

5.1 Zadanie 1 – zliczanie wartości parzystych 44
5.2 Zadanie 2 – obliczanie iloczynu elementów listy 46
5.3 Zadanie 3 – obliczanie średniej nieparzystych elementów 47
5.4 Zadanie 4 – zmiana wartości elementów na liście 48
5.5 Zadanie 5 – usuwanie wartości 49
5.6 Zadanie 6 – wyszukiwanie powtarzających się nazwisk 50
5.7 Zadanie 7 – znajdowanie czynników pierwszych liczby 53

6 Krotki 56

6.1 Zadanie 1 – wyznaczanie obiektów w zasięgu radaru 56
6.2 Zadanie 2 – wyszukiwanie najwyższego drzewa 58
6.3 Zadanie 3 – wyszukiwanie najwyższego drzewa raz jeszcze 59

7 Słowniki 61

7.1 Zadanie 1 – wyznaczanie częstotliwości liter w tekście 61
7.2 Zadanie 2 – wyszukiwanie drzew danego gatunku w lesie 64
7.3 Zadanie 3 – wyznaczanie dzielników właściwych liczby 66

8 Funkcje 68

8.1 Zadanie 1 – obliczanie iloczynu dwóch liczb 68
8.2 Zadanie 2 – obliczanie ilorazu dwóch liczb 69
8.3 Zadanie 3 – obliczanie sumy elementów listy 71
8.4 Zadanie 4 – sprawdzanie, czy liczba jest liczbą pierwszą 71
8.5 Zadanie 5 – wyznaczanie liczb pierwszych z przedziału 73
8.6 Zadanie 6 – wyznaczanie liczb bliźniaczych z przedziału 74
8.7 Zadanie 7 – wyznaczanie liczb zaprzyjaźnionych z przedziału . . . 75

9 Pliki 79

9.1 Zadanie 1 – rozdzielenie wartości 81
9.2 Zadanie 2 – wyszukiwanie towarów o największej podwyżce cen 83
9.3 Zadanie 3 – podsumowanie ocen 85

Wstęp

Początki języka Python (https://www.python.org/) sięgają lat 90. XX wieku. Jego twórcą jest holenderski programista Guido van Rossum, który zaczerpnął nazwę swojego projektu od „Latającego cyrku Monty Pythona", którego był wielkim fanem. W owym czasie wśród najpopularniejszych języków były m.in. C++ oraz Perl, które można uznać za pierwowzory powstania języka Python, czerpiącego z nich to, co najbardziej atrakcyjne. C++ stanowił przykład języka obiektowego i elastycznego w sensie stylu programowania, natomiast Perl był językiem skryptowym, dynamicznie typowanym i dostępnym na wiele systemów operacyjnych.
Język Python jest obecnie szeroko stosowany zarówno do tworzenia serwisów internetowych, jak i aplikacji desktopowych, tworzenia interfejsów użytkownika, programowania aplikacji sieciowych, gier etc. Rozwijane są liczne frameworki, m.in. Django, Pyramids, wspomagające tworzenie zaawansowanych aplikacji z powodzeniem stosowanych w komercyjnych rozwiązaniach. Jednakże w ostatnich latach jego rosnąca popularność związana jest głównie z zastosowaniami w analizie danych oraz uczeniu maszynowym, dzięki udostępnianym licznym bibliotekom, takim jak NumPy, Pandas, SciPy, TensorFlow, Matplotlib oraz wielu innym. Analizując rankingi popularności poszczególnych języków w określonych dziedzinach zastosowań, można nawet zaryzykować stwierdzenie, że język Python stał się podstawowym narzędziem naukowców i specjalistów do tworzenia tego rodzaju aplikacji.
Język Python jest udostępniany na licencji Open Source, a dokładnie Python Software Foundation License (PSFL), będącą liberalną licencją oprogramowania w stylu BSD. Można go swobodnie używać i rozpowszechniać, nawet do użytku komercyjnego, a jego filozofia (The Zen of Python) jest przedstawiana na oficjalnych stronach jako:

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.

Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one - and preferably only one - obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Niniejszy skrypt jest przeznaczony, przede wszystkim dla studentów kierunku *Matematyka stosowana* prowadzonego na Wydziale Informatyki w ramach przedmiotu *Podstawy programowania* oraz jako narzędzie wspomagające przygotowanie do realizacji zadań praktycznych w ramach wielu innych przedmiotów, takich jak *Programowanie obiektowe*, *Algorytmy i struktury danych* oraz *Sztuczna inteligencja*.
Jeśli w tekście publikacji znajdziecie błąd lub zechcecie podzielić się jakąś sugestią czy pomysłem odnośnie do zadań bądź rozwiązań – zachęcamy do kontaktu. Nasze adresy e-mail znajdziecie na stronie Wydziału Informatyki: https://wi.pb.edu.pl/pracownicy/lista-pracownikow/. Życzymy miłej lektury i sukcesów w programowaniu.

Białystok, maj 2023 r.

Anna Łupińska-Dubicka
Andrzej Chmielewski

Rozdział 1

Instrukcje wejścia–wyjścia

Do wyświetlania danych na ekranie służy funkcja `print`.

```
print("Good morning!")
```

Funkcja `print` wyświetla na ekranie przekazany w parametrze tekst.

Do wprowadzania danych służy funkcja `input`.

```
name = input("What's your name?")
print("Hello", name)
```

Funkcja `input` przyjmuje jako parametr wyświetlany na ekranie napis. Możemy ją również wywołać bez parametru: `input()`, jednakże w takiej sytuacji warto by było wyświetlić wcześniej informację, co chcemy wczytać od użytkownika. Funkcja `input` wczytuje z klawiatury ciąg znaków i umieszcza go w zmiennej (w naszym przykładzie w zmiennej `name`). Należy pamiętać, że wczytane wartości są zawsze traktowane jako napis.

Zmienna jest to nazwane miejsce w pamięci komputera, w którym możemy przechowywać pewną wartość. *Python* pozwala na używanie zmiennych do przechowywania wartości dowolnego typu.

Jak widzimy w powyższym przykładzie, jeśli chcemy wypisać na ekranie więcej niż jeden element, to kolejne wartości oddzielamy przecinkami. Domyślnie funkcja `print` przechodzi do nowego wiersza. Jeśli chcemy tego uniknąć, możemy wywołać ją z parametrem `end`. Określa on, czym kończy się wyświetlany ciąg.

Stwórzmy dwie zmienne i nadajmy im wartości:

```
a = 5
b = 6
```

Ponieważ w *Pythonie* możliwe jest jednoczesne nadanie wartości kilku zmiennym, powyższe dwie instrukcje możemy zapisać w skróconej postaci. Zapis:

```
a, b = 5, 6
```

przypisze do zmiennej `a` wartość 5, a do zmiennej `b` wartość 6.

Zmienne `a` i `b` wyświetlą się w jednym wierszu:

```
print(a, b)
```

Zmienne `a` i `b` wyświetlą się w dwóch wierszach:

```
print(a)
print(b)
```

Zmienne `a` i `b` wyświetlą się w jednym wierszu ze znakiem spacji pomiędzy nimi:

```
print(a, end=" ")
print(b)
```

Zmienne `a` i `b` wyświetlą się w jednym wierszu ze średnikiem i znakiem spacji pomiędzy nimi:

```
print(a, end="; ")
print(b)
```

1.1 Zadanie 1 – obliczanie średniej dwóch liczb

Napisz program wyświetlający na ekranie średnią arytmetyczną dwóch wartości podanych przez użytkownika.

Zaczynamy od wczytania liczb:

```
a = input("Enter first number: ")
b = input("Enter second number: ")
```

W języku *Python* nie ma potrzeby deklaracji zmiennej. Zmienna jest tworzona dopiero w momencie, gdy przypiszemy do niej wartość. Każda wartość jest jakiegoś typu, np. wartość 10 to liczba całkowita, 0.75 to liczba rzeczywista, a *"Ala"* to tekst. Przy tworzeniu zmiennej nie trzeba podawać typu danych, które będą w niej przechowywane. *Python* automatycznie określa typ danych na podstawie wartości, którą podaliśmy. Zmienna przyjmuje taki typ wartości, jaka jest w niej przechowywana. W każdej chwili do zmiennej możemy przypisać wartość innego typu.
Następnie wyznaczamy średnią i wypisujemy ją na ekranie:

```
avg = (a+b)/2
print("Average = ", avg)
```

Przy próbie uruchomienia programu otrzymamy następujący błąd: *unsupported operand type(s) for /: 'str' and 'int'.*

Co się stało? Jak wspomniano wcześniej, funkcja `input` traktuje wczytane

wartości jako napis. Aby móc wykonywać operacje matematyczne musimy najpierw przekształcić je do postaci liczbowej:

```
a = float(a)
b = float(b)
```

Funkcja `float()` konwertuje wczytany napis na liczbę rzeczywistą. Podobne działanie ma funkcja `int()` – przekształca pobrany ciąg na liczbę całkowitą. Oczywiście, konwersja jest możliwa tylko w przypadku, gdy użytkownik wpisze właściwe znaki: cyfry oraz, w przypadku wartości rzeczywistych, `'.'` jako separator dziesiętny. W przeciwnym razie program nadal będzie zgłaszał błąd. Jak sobie z tym radzić, dowiesz się w dalszej części skryptu.
Konwersję możemy przeprowadzić od razu w momencie wczytywania wartości:

```
a = float(input("Enter first number: "))
b = float(input("Enter second number: "))
```

Cały program wyglądać może zatem następująco:

```
a = float(input("Enter first number: "))
b = float(input("Enter second number: "))
avg = (a+b)/2
print("Average = ", avg)
```

1.2 Zadanie 2 – wyznaczanie pola kwadratu

Napisz program, który wyświetla wartość pola kwadratu o podanym przez użytkownika boku.

```
a=float(input("podaj bok kwadratu a: "))
pole = a*a
print("pole kwadratu o boku: ", a, "wynosi: ", pole)
```

W momencie, gdy do wypisania mamy więcej zmiennych, budowanie napisu w funkcji `print` może być uciążliwe. Możemy wtedy skorzystać z alternatywnej metody:

```
print(f"pole kwadratu o boku: {a} wynosi: {pole}")
```

Litera `f` (można również stosować wielką literę `F`) wewnątrz wywołania funkcji `print` mówi, że w miejscu każdej pary nawiasów klamrowych `{}` ma się znaleźć wartość zmiennej podanej w tych nawiasach, czyli zapis `{a}` zostanie przy wyświetlaniu zastąpiony długością boku podaną przez użytkownika.
Możemy dodatkowo podać liczbę miejsc po przecinku wyświetlanych na ekranie:

```
print(f"pole kwadratu o boku: {a:.2f} wynosi: {pole:.4f}")
```

W tym wypadku zmienna `a` zostanie wyświetlona z dokładnością dwóch miejsc po przecinku, a zmienna `pole` do czterech.

1.3 Zadanie 3 – symulacja rzutu kostką

Napisz program symulujący rzut sześciościenną kostką i wyświetlający na ekranie wynik rzutu.

Aby wylosować wartość, potrzebujemy funkcji losującej. *Python* oferuje nam kilka takich funkcji, w zależności od tego, co chcemy osiągnąć. Zanim skorzystamy z jakiejkolwiek z nich, musimy wskazać moduł, w którym one się znajdują. Moduły są po prostu zbiorem pewnych funkcjonalności oferowanych przez dany język programowania (w innych językach noszą one nazwę bibliotek lub pakietów). Moduły importujemy do swojego programu za pomocą komendy `import`. Do rozwiązania naszego zadania wykorzystamy moduł `random`:

```
import random
```

Pamiętaj, że każdy moduł, z którego będziemy korzystać, powinien być zaimportowany, zanim odwołamy się do jego zawartości. Dla zachowania porządku w kodzie najlepiej jest to robić na początku pliku z programem.

Losowanie liczby całkowitej realizowane jest za pomocą funkcji `random.randint()`. Wymaga ona dwóch parametrów – początku oraz końca przedziału (przedział jest obustronnie domknięty):

```
diceThrow = int(random.randint(1,6))
```

Kod całego programu może wyglądać zatem następująco:

```
import random
diceThrow = int(random.randint(1,6))
print(f"wylosowano: {diceThrow}")
```

Rozdział 2

Instrukcje warunkowe

Instrukcja warunkowa pozwala wykonywać pewne instrukcje przy spełnionym bądź niespełnionym warunku. Dzięki niej możliwe jest rozgałęzienie w działaniu programu – w zależności od zastanej sytuacji może on działać w różny sposób.

2.1 Zadanie 1 – sprawdzanie parzystości liczby

Napisz program wypisujący na ekranie komunikat w przypadku, gdy wpisana liczba całkowita jest liczbą parzystą.

Zaczynamy od pobrania od użytkownika wartości, jednocześnie konwertując wczytany tekst na liczbę:

```
number = int(input("Enter a number"))
```

Do sprawdzenia parzystości liczby wykorzystamy instrukcję warunkową. Pozwala ona na zaznaczenie bloku instrukcji, który się wykona wyłącznie wtedy, gdy prawdziwe będzie wyrażenie logiczne sprawdzane przed wejściem do tego bloku. Jeśli warunek jest spełniony, wykonywany jest blok instrukcji. W naszym przykładzie instrukcja warunkowa ma następującą postać:

```
if number%2 == 0:
    print("f{number} is even")
```

Składa się ze słowa kluczowego `if`, po którym następuje sprawdzany warunek: `number%2 == 0`. Podwójny znak równości `==` określa równość dwóch wartości. Po warunku pojawia się znak `:` (dwukropek) oznaczający początek bloku instrukcji.
Zwróć uwagę, że linia zawierająca wywołanie `print` jest wcięta w prawo. W języku *Python* oznacza to, że jest ona zależna od znajdującej się wyżej od niej linii kodu zakończonej dwukropkiem (`if`). Jeśli usuniemy to wcięcie, program przestanie działać. Tam, gdzie kończy się wcięcie, kończy się nasz blok warunkowy.

Pełny kod przykładu wygląda następująco:

```
number = int(input("Enter a number"))
if number%2 == 0:
    print("f{number} is even")
```

Inne podstawowe operatory porównania to: nierówne `!=`, mniejsze `<` i większe `>` oraz mniejsze lub równe `<=` i większe lub równe `>=`. Wykorzystywane są również operatory logiczne: `not` – negacja, `and` – koniunkcja oraz `or`, czyli alternatywa.

2.2 Zadanie 2 – sprawdzanie, czy rok jest przestępny

Napisz program wczytujący z klawiatury liczbę całkowitą reprezentującą rok, a następnie wypisujący informację o tym, czy jest lub nie jest to rok przestępny.

Na początku wczytujemy od użytkownika wartość zmiennej `year`, pamiętając o konwersji na liczbę całkowitą:

```
year = int(input("Enter a year"))
```

Następnie tworzymy warunek sprawdzający, czy wpisana wartość reprezentuje rok przestępny:

```
if year%4 == 0 and year%100 != 0 or year%400 == 0:
    print(f"{year} is a leap year.")
```

Używamy warunku złożonego, korzystając z operatorów koniunkcji (`and`) i alternatywy (`or`). Tak zapisany warunek odczytujemy następująco: reszta z dzielenia przez 4 równa 0 i jednocześnie reszta z dzielenia przez 100 różna od 0 lub reszta z dzielenia przez 400 równa 0. W przypadku koniunkcji wszystkie jej elementy muszą być prawdziwe, w przypadku alternatywy wystarczy jeden prawdziwy. Zatem zmienna `year` jednocześnie nie może dzielić się przez 4 i przez 100. Należy pamiętać, że operator koniunkcji ma wyższy priorytet niż operator alternatywy (analogicznie jak operatory mnożenia i sumowania).
W przypadku, gdy chcemy również zareagować na niespełnienie warunku, możemy rozbudować instrukcję warunkową o blok `else` (interpretowane jako *w przeciwnym przypadku*).

```
if year%4 == 0 and year%100 != 0 or year%400 == 0:
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

Instrukcje w bloku `else` wykonują się w sytuacji, gdy warunek zawarty w `if` nie został spełniony.

Cały program wygląda zatem następująco:

```
year = int(input("Enter a year"))
```

```
if year%4 == 0 and year%100 != 0 or year%400 == 0:
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

2.3 Zadanie 3 – prosta w układzie współrzędnych

Napisz program, który po wczytaniu od użytkownika a i b współczynników prostej wyświetli na ekranie numery ćwiartek układu współrzędnych, przez które ona przechodzi.

Zaczynamy od pobrania wartości od użytkownika:

```
a = float(input("Enter a: "))
b = float(input("Enter b: "))
```

Następnie przystępujemy do rozpatrzenia możliwych wartości zmiennej `a`. Musimy uwzględnić trzy przypadki: $a > 0$, $a < 0$ oraz $a = 0$.

```
if a>0:
{
}
elif a<0:
{
}
else:
{
}
```

Powyższy kod możemy odczytać w następujący sposób: jeżeli $a > 0$ ... *w przeciwnym wypadku, jeżeli* $a < 0$ ... *w przeciwnym wypadku* ... Nowy element `elif` rozbudowuje instrukcję warunkową o możliwość uwzględnienia więcej niż dwóch możliwych warunków. Liczba zastosowanych instrukcji `elif` zależy tylko od rozpatrywanego problemu – może być jedna, dwie albo więcej.
W analogiczny sposób, jako zagnieżdżone instrukcje warunkowe, przeanalizujemy wartości zmiennej `b`:

```
if a>0:
    if b>0:
        print("goes through: I, II, III.")
    elif b<0:
        print("goes through: I, III, IV.")
    else:
        print("goes through: I, III.")
```

Pełny kod programu wyglądać będzie następująco:

```
a = float(input("Enter a: "))
b = float(input("Enter b: "))

print(f"The line with the equation {a}x+{b}f=0", end=" ")
if a>0:
    if b>0:
        print("goes through: I, II, III.")
    elif b<0:
        print("goes through: I, III, IV.")
    else:
        print("goes through: I, III.")
elif a<0:
    if b>0:
        print("goes through: I, II, IV.")
    elif b<0:
        print("goes through: II, III, IV.")
    else:
        print("goes through: II, IV.")
else:    #a==0
    if b>0:
        print("goes through: I, II.")
    elif b<0:
        print("goes through: III, IV.")
    else:
        print("coincides with the OX axis.")
```

2.4 Zadanie 4 – wystawianie oceny

Napisz program, który wystawi studentowi ocenę z kolokwium. Użytkownik podaje liczbę uzyskanych punktów wg schematu:

$< 0 - 50 >$	2.0
$(50 - 60 >$	3.0
$(60 - 70 >$	3.5
$(70 - 80 >$	4.0
$(80 - 90 >$	4.5
$(90 - 100 >$	5.0

Pamiętaj o sprawdzeniu, czy podana wartość mieści się w przedziale $< 0; 100 >$.

Po wczytaniu liczby uzyskanych punktów:

```
points = int(input("Enter number of points:"))
```

przystępujemy do tworzenia instrukcji warunkowej. Każda ocena zależy od pewnego przedziału punktowego. Przykładowo, dla oceny 2.0 warunek możemy zapisać następująco:

```
if points>=0 and points<=50:
    print("You got 2.0")
```

Ale *Python* umożliwia nam również taki zapis:

```
if 0<=points<=50:
    print("You got 2.0")
```

W analogiczny sposób tworzymy pozostałe warunki:

```
if 0<=points<=50:
    print("You got 2.0")
elif 50<points<=60:
    print("You got 3.0")
elif 60<points<=70:
    print("You got 3.5")
elif 70<points<=80:
    print("You got 4.0")
elif 80<points<=90:
    print("You got 4.5")
elif 90<points<=100:
    print("You got 5.0")
```

W powyższym kodzie możemy zrezygnować z warunków złożonych w sekcjach `elif`, ponieważ wykluczają się one nawzajem. Przykładowo, jeżeli zmienna `points` przyjmuje wartość 64, to pierwsze dwa warunki nie zostaną spełnione i wystarczy sprawdzić, czy jest ona mniejsza lub równa 70. Taki zapis jednak wymaga od nas zachowania konkretnej kolejności – sprawdzane wartości graniczne muszą być posortowane rosnąco:

```
if 0<=points<=50:
    print("You got 2.0")
elif points<=60:
    print("You got 3.0")
elif points<=70:
    print("You got 3.5")
elif points<=80:
    print("You got 4.0")
```

```
elif points<=90:
    print("You got 4.5")
elif points<=100:
    print("You got 5.0")
```

Ale co w przypadku, gdy wprowadzimy złą liczbę punktów? Przy obecnym kodzie na ekranie nie pojawi się żaden komunikat. Dodajmy zatem blok `else`, który obsłuży nam sytuację punktów spoza skali:

```
else:
    print("Wrong number of points")
```

Pełny kod programu może wyglądać następująco:

```
points = int(input("Enter number of points:"))
if 0<=points<=50:
    print("You got 2.0")
elif points<=60:
    print("You got 3.0")
elif points<=70:
    print("You got 3.5")
elif points<=80:
    print("You got 4.0")
elif points<=90:
    print("You got 4.5")
elif points<=100:
    print("You got 5.0")
else:
    print("Wrong number of points")
```

Rozdział 3

Instrukcje iteracyjne

Instrukcja iteracyjna (zwana również pętlą) polega na powtarzaniu pewnego ciągu instrukcji skończoną liczbę razy. W języku *Python* są dostępne dwie instrukcje iteracyjne: `for` oraz `while`.

Pętla `for`

Pętla `for` używana jest do przetworzenia elementów pewnej sekwencji:

```
for zmienna in sekwencja:
               instrukcje_do_powtórzenia
```

gdzie:

- `zmienna` to zmienna, która przechowuje pojedynczy element pobrany z sekwencji w kolejnych przebiegach pętli,
- `sekwencja` to sekwencja, z której pobierane są kolejne wartości i wstawiane do zmiennej,
- `instrukcje_do_powtórzenia` to polecenia, które mają być powtórzone w każdym obiegu pętli.

Przetestuj poniższy fragment kodu:

```
for x in [2, 4, 6, 8, 10]:
    print(x)
```

W powyższym fragmencie kodu pętla `for` wykona się 5 razy, ponieważ sekwencja [2, 4, 6, 8, 10] liczy 5 elementów. Zwróć uwagę, że elementy sekwencji zapisywane są w nawiasach kwadratowych. W każdej iteracji do zmiennej lokalnej `x` zostanie przypisana wartość kolejnego elementu sekwencji, a następnie wyświetlona na ekranie.
Sekwencję możemy również przypisać do zmiennej:

```
data = [2, 4, 6, 8, 10]
for x in data:
    print(x)
```

Jak widzisz, pętla `for` *koncentruje* się na kolejnych wartościach sekwencji. Jeżeli poza wartością następnego elementu w sekwencji potrzebna jest nam również jego pozycja w sekwencji, możemy użyć funkcji `enumerate()`:

```
data = [1, 4, 3, 5, 9, 5, 8, 8, 2, 10]
for index, x in enumerate(data):
    if x%2 == 0:
        print(f"Number {x} at the position {index+1} is even")
```

Funkcja `enumerate()` zwraca nam parę *pozycja–wartość* i przypisuje ją do zmiennych `index` oraz `x`. Pozycje numerowane są od 0, stąd wyrażenie `index+1` wewnątrz funkcji `print`.

Nie zawsze mamy możliwość/potrzebę przetwarzania danych pobieranych z istniejącej sekwencji. W takiej sytuacji sekwencję użytą w pętli tworzymy *ad hoc* przy użyciu funkcji `range`.

```
for x in range{5}:
    print("*")
```

Powyższy fragment kodu wyświetli na ekranie 5 symboli `'*'`. Niejawnie zostanie wygenerowana sekwencja `[0, 1, 2, 3, 4]`, po której iteruje pętla `for`.

Funkcja `range` może być wywoływana z trzema parametrami:

```
range(start, stop, step)
```

gdzie:

- `start` wyznacza początkową wartość w sekwencji (domyślnie wartość 0),
- `stop` wyznacza wartość końcową, która nie jest zwracana w wyniku (parametr wymagany),
- `step` określa różnicę pomiędzy elementami sekwencji, może być wartością dodatnią lub ujemną – w takim wypadku $start > stop$ (domyślnie wartość 1).

Wszystkie parametry muszą być liczbami całkowitymi!

Kod wyświetlający na ekranie liczby od 0 do 9:

```
for x in range(10):
    print(x, end=" ")
```

Kod wyświetlający na ekranie liczby od 1 do 9:

```
for x in range(1, 10):
    print(x, end=" ")
```

Kod wyświetlający na ekranie liczby $1, 3, 5, 7, 9$:

```
for x in range(1, 10, 2):
    print(x, end=" ")
```

Kod wyświetlający na ekranie liczby $10, 8, 6, 4, 2$:

```
for x in range(10, 1, -2):
    print(x, end=" ")
```

Pętla `while`

Drugą pętlą dostępną w języku *Python* jest pętla `while`. Wykonuje się ona, dopóki pewien warunek logiczny jest spełniony:

```
while warunek:
    instrukcje_do_powtórzenia
```

Kod wyświetlający na ekranie liczby od 0 do 9:

```
x = 0
while x<10:
    print(x, end=" ")
    x += 1
```

Kod wyświetlający na ekranie liczby od 1 do 9:

```
x = 1
while x<10:
    print(x, end=" ")
    x += 1
```

Kod wyświetlający na ekranie liczby $1, 3, 5, 7, 9$:

```
x = 1
while x<10:
    print(x, end=" ")
    x += 2
```

Kod wyświetlający na ekranie liczby $10, 8, 6, 4, 2$:

```
x = 10
while x>1:
    print(x, end=" ")
    x -= 2
```

Kod pobierający liczbę punktów tak długo, aż student uzyska wymaganą do zaliczenia liczbę punktów:

```
points = int(input("Enter your points: "))
while points<=50:
    print("You haven't passed")
    points = int(input("Enter your points again: "))
print("You have passed the test!")
```

W przypadku pętli `while` musimy pamiętać o nadaniu wartości początkowej zmiennej sterującej (czyli tej, którą sprawdzamy w warunku) oraz o zmianie jej wartości w każdym przebiegu pętli.

Instrukcja `break`

Wykonanie instrukcji `break` powoduje przerwanie wykonywania najściślej otaczającej ją pętli. Może być stosowane w pętli `while`, a także w pętli `for`.

Poniżej znajduje się kod wczytujący co najwyżej 10 liczb i przerywający wczytywanie, gdy użytkownik poda wartość ujemną.

```
counter = 0
for i in range(10):
    x = int(input("Enter number"))
    if x<0:
        print("Wrong number!")
        break
    counter += 1
print(f"You entered {counter} valid numbers.")
```

Czasami trudno jest określić warunek zakończenia pętli. Możemy w takim wypadku użyć tzw. *pętli nieskończonej* i opuścić ją, korzystając z instrukcji `break`.

Kod wczytujący liczby od użytkownika dopóki nie zostanie wprowadzona wartość nieparzysta:

```
counter = 0
while True:
    x = int(input("Enter number"))
    if x%2 == 1:
        print("Odd number!")
        break
    counter += 1
print(f"You entered {counter} even numbers.")
```

Instrukcja `continue`

Za pomocą instrukcji `continue` przechodzimy do kolejnej iteracji pętli. Możemy jej używać zarówno w pętli `for`, jak i `while`.

Kod wczytujący od użytkownika 10 liczb parzystych:

```
counter = 0
for i in range(10):
    x = int(input("Enter number"))
    if x%2 == 0:
        print("Even number!")
        continue
    counter += 1
print(f"You entered {counter} odd numbers.")
```

Zwróć uwagę, że zwiększenie zmiennej `counter` umieszczone zostało poniżej instrukcji warunkowej z `continue`. Gwarantuje to nam, że w przypadku wprowadzenia wartości nieparzystej nie zostanie ona wliczona do wprowadzanych wartości. Dzięki temu pętla zatrzyma się dopiero po podaniu wymaganych 10 liczb parzystych, bez względu ile nieprawidłowych wartości w międzyczasie wprowadził użytkownik.

Instrukcja `else`

W języku *Python* za blokiem kodu zawierającym pętlę możemy użyć instrukcji `else`. W takim wypadku, w momencie gdy warunek w pętli staje się fałszywy, wykonane zostaną instrukcje umieszczone w bloku `else`. Jednakże, gdy opuścimy pętlę instrukcją `break`, sekcja `else` zostanie pominięta.

```
for i in range(5):
    x = int(input(f"Enter {i+1} odd number: "))
    if (x%2 == 0):
        print(f"{x} is an even number!")
        break
else:
    print("You know the difference between odd and even numbers.")
```

Komunikat *You know the difference between odd and even numbers.* pojawi się na ekranie tylko wtedy, gdy pętla wykona się dokładnie 5 razy, czyli kiedy wszystkie wprowadzone wartości będą parzyste. W przypadku, gdy podana

liczba będzie nieparzysta, pętla zostanie przerwana instrukcją `break`, blok `else` zostanie zignorowany.

3.1 Zadanie 1 – zliczanie parzystych

Napisz program, który wczyta od użytkownika 10 liczb i zliczy, ile z nich jest parzystych. Wartość n podawana jest przez użytkownika na początku działania programu.

Po wyświetleniu informacji dla użytkownika:

```
print("Enter 10 numbers: ")
```

tworzymy i zerujemy wartość licznika (zmienna `counter`):

```
counter = 0
```

a następnie piszemy nagłówek instrukcji iteracyjnej:

```
for i in range(10):
```

Zmienna `i` jest zmienną sterującą i przejmować będzie kolejne wartości całkowite z przedziału $< 0, 10)$. W każdym przebiegu pętli musimy wczytać wartość od użytkownika (pamiętaj, że wczytywane wartości są napisami i należy je przekonwertować na typ całkowity):

```
x = int(input("Enter number"))
```

Po wczytaniu wartości sprawdzamy, czy jest ona parzysta. Korzystamy z operatora *modulo* zapisywanego znakiem %. Zwraca on resztę z dzielenia – liczba parzysta podzielona przez 2 ma resztę równą 0. Jeżeli ten warunek jest spełniony, to zwiększamy wartość licznika `counter` o jeden:

```
if x%2 == 0:
    counter += 1
```

Po zakończeniu pętli wyświetlamy na ekranie komunikat o liczbie parzystych wartości:

```
if counter == 0:
    print("There is no even numbers.")
elif counter == 1:
    print("There is 1 even number.")
else:
    print(f"There are {counter} even numbers.")
```

Dla czytelności działania uwzględniamy różne warianty zmiennej `counter`.

Kod całego programu prezentuje się następująco:

```
print("Enter 10 numbers: ")
counter = 0
for i in range(10):
    x = int(input("Enter number"))
    if x%2 == 0:
        counter += 1

if counter == 0:
    print("There is no even numbers.")
elif counter == 1:
    print("There is 1 even number.")
else:
    print(f"There are {counter} even numbers.")
```

3.2 Zadanie 2 – suma poprzedników

Napisz program, który dla podanej przez użytkownika liczby całkowitej obliczy i wyświetli sumę jej poprzedników.
Na przykład dla wartości 6 *wynikiem jest* $1 + 2 + 3 + 4 + 5 = 15$

Zaczynamy od wyświetlenia komentarza oraz wczytania wartości:

```
number = int(input("Enter a number"))
```

Następnie tworzymy i zerujemy zmienną, w której będziemy przechowywać sumę poprzedników wczytanej liczby:

```
sum = 0
```

i konstruujemy pętlę:

```
for i in range(1, number):
    sum += i
```

Funkcja `range(1, number)` generuje ciąg liczb w postaci: $1, 2, number - 1$. Te wartości trafią po kolei do zmiennej sterującej `i`. W każdej iteracji zmienna `sum` zostanie zwiększona o bieżącą wartość `i`.
Po zakończeniu pętli wyświetlamy na ekranie obliczony wynik:

```
print(f"Sum of number less than {number} equals {sum}")
```

Pełny kod programu wygląda zatem następująco:

```
number = int(input("Enter a number"))
sum = 0
for i in range(1, number):
    sum += i
print(f"Sum of number less than {number} equals {sum}")
```

3.3 Zadanie 3 – suma liczb nieparzystych z przedziału

Napisz program, który zsumuje wartości nieparzyste z podanego przez użytkownika przedziału.

Zaczynamy od pobrania od użytkownika początku i końca przedziału:

```
start = int(input("Enter start"))
stop = int(input("Enter stop"))
```

Powinniśmy wziąć pod uwagę przypadek, gdy użytkownik poda początek przedziału większy od jego końca. W takiej sytuacji moglibyśmy oczywiście nic nie liczyć, ale możemy również naprawić jego błąd, zamieniając wartościami zmienne `start` i `stop`:

```
if start>stop:
    start, stop = stop, start
```

Jak pamiętasz, *Python* umożliwia jednoczesne przypisanie wartości kilku zmiennym. Zatem zapis `start, stop = stop, start` wstawi do zmiennej `start` dotychczasową wartość zmiennej `stop`, a do zmiennej `stop` trafi dotychczasowa wartość zmiennej `start`.

Następnie tworzymy i zerujemy zmienną, w której będziemy trzymać sumę elementów:

```
sum = 0
```

i rozpoczynamy instrukcję iteracyjną:

```
for x in range(start, stop+1):
```

Tym razem skorzystaliśmy z funkcji `range`, podając początek i koniec przedziału generowanych wartości. Zapis `stop+1` gwarantuje nam uwzględnienie końca przedziału jako ostatniej rozpatrywanej wartości.
Pozostało tylko sprawdzić nieparzystość liczby i powiększenie sumy o jej wartość:

```
    if x%2 == 1:
        sum += x
```

Pełny kod programu może wyglądać następująco:

```
start = int(input("Enter start"))
stop = int(input("Enter stop"))
if start>stop:
    start, stop = stop, start
sum = 0
for x in range(start, stop+1):
    if x%2 == 1:
        sum += x
print(f"Sum of odd numbers equals {sum}")
```

Te zadanie można rozwiązać również w inny sposób wykorzystując trzeci parametr funkcji `range`.

```
for x in range(start, stop+1, 2):
    if x%2 == 1:
        sum += x
```

W tym wypadku musimy jednakże sprawdzić, czy początek przedziału jest liczbą nieparzystą. Jeżeli nie, to końcowa wartość zmiennej `sum` będzie wynosiła 0. Dlaczego? Załóżmy, że `stop = 4, stop = 10`. Krok równy 2 w funkcji `range` spowoduje, że wygenerowane liczby w sekwencji będą równe: $4, 6, 8, 10$. Żadna z tych wartości nie jest nieparzysta, więc nie zostanie uwzględniona w sumowaniu.
Aby naprawić ten błąd, musimy zadbać, by pierwsza wartość wygenerowana przez `range` była nieparzysta:

```
if start%2 == 0:
    start += 1
```

W przypadku, gdy podany przez użytkownika przedział zaczyna się od wartości parzystej, zwiększamy jego początek o 1, na wartość nieparzystą. Oczywiście, robimy to tylko w przypadku, gdy `start` jest parzysta. Jeżeli przedział zaczyna się od liczby nieparzystej, nic nie zmieniamy.

Po tych zmianach program będzie wyglądał następująco:

```
start = int(input("Enter start"))
stop = int(input("Enter stop"))
if start>stop:
    start, stop = stop, start
```

```
sum = 0
if start%2 == 0:
    start += 1
for x in range(start, stop+1, 2):
    print(x)
    if x%2 == 1:
        sum += x
print(f"Sum of odd numbers equals {sum}")
```

3.4 Zadanie 4 – suma mniejsza od 100

Napisz program, który będzie wczytywał liczby, dopóki ich suma będzie mniejsza od 100. Po zakończeniu wczytywania wyświetli, ile liczb zostało wprowadzonych przed osiągnięciem granicy 100.

W tym zadaniu użyjemy dwóch zmiennych: `sum` do obliczania sumy oraz `counter` do zliczania wprowadzanych wartości:

```
sum = 0
counter = 0
```

Ponieważ nie jesteśmy w stanie określić liczby iteracji, wykorzystamy pętlę `while`. Będzie się ona wykonywać tak długo, aż wartość sumy osiągnie lub przekroczy 100:

```
while sum<100:
```

Wewnątrz pętli wczytujemy od użytkownika liczbę oraz zwiększamy wartość sumy i licznika:

```
x = int(input("Enter number:"))
counter += 1
sum += x
```

Na zakończenie wypisujemy jeszcze, ile liczb zostało wprowadzonych:

```
print(f"You entered {counter} values.")
```

Kod całego programu wygląda zatem następująco:

```
sum = 0
counter = 0
while sum<100:
    x = int(input("Enter number:"))
    counter += 1
```

```
    sum += x
print(f"You entered {counter} values.")
```

3.5 Zadanie 5 – podaj hasło

Napisz program, który sprawdza zgodność wprowadzonego hasła z zapisanym wzorcem. Użytkownik ma trzy próby na podanie właściwego ciągu znaków.

Zaczniemy od umieszczenia w kodzie hasła, z którym będziemy porównywać wartości prowadzone przez użytkownika:

```
pattern = "S3cret_C0de"
```

Jak widzisz, zmienna `pattern` przechowuje ciąg znaków. W *Pythonie* ten typ danych nosi nazwę napisów bądź łańcuchów. Zawartość napisu otoczona jest apostrofami ' lub cudzysłowami ". Ponieważ znamy liczbę iteracji, możemy użyć pętli `for`. Oczywiście, te zadanie można również rozwiązać, korzystając z pętli `while`.

```
for i in range(3):
```

Pierwszą instrukcją pętli jest wczytanie od użytkownika jego hasła:

```
    password = input(f"Trial n {i+1}")
```

Zwróć uwagę, że tym razem wywołujemy funkcję `input`, nie wykonując dodatkowego rzutowania, jak we wcześniejszych zadaniach. Funkcja `input` zwraca wczytaną wartość w postaci napisu, a to jest to, czego potrzebujemy.

Następnie sprawdzamy, czy wprowadzony ciąg znaków jest identyczny z zapisanym wzorcem. Jeżeli tak, wyświetlamy stosowny komunikat i opuszczamy pętlę:

```
    if password == pattern:
        print("Access granted")
        break
```

Napisy porównujemy dokładnie w taki sam sposób jak zmienne liczbowe. Dwa napisy są sobie równe wtedy i tylko wtedy, gdy składają się z identycznych znaków umieszczonych na identycznych pozycjach. Wielkość liter jest istotna.

Pętla `for` zakończy się po wprowadzeniu maksymalnie trzech ciągów. Jeżeli w tym czasie użytkownik nie poda prawidłowego hasła, na ekranie pojawi się komunikat *Permission denied* wypisywany w bloku `else` pętli `for`:

```
else:
    print("Permission denied")
```

Dodatkowo, bez względu na wynik logowania, program życzy użytkownikowi miłego dnia:

```
print("Have a nice day");
```

Pełny kod programu może wyglądać następująco:

```
pattern = "S3cret_C0de"

print("Enter password. You have 3 trials!")
for i in range(3):
    password = input(f"Trial n {i+1}")
    if password == pattern:
        print("Access granted")
        break
else:
    print("Permission denied")
print("Have a nice day");
```

3.6 Zadanie 6 – zgadnij liczbę

Napisz program, który pyta się o pewną wylosowaną przez program liczbę z przedziału określonego przez użytkownika tak długo, aż zostanie odgadnięta. Po każdej nieudanej próbie informuje, czy szukana liczba jest większa, czy mniejsza od podanej.

Zaczynamy od zaimportowania modułu z funkcją losującą:

```
import random
```

i wczytania krańców przedziału, z którego program wylosuje liczbę:

```
start = int(input("Enter start"))
stop = int(input("Enter stop"))
if start>stop:
    start, stop = stop, start
```

Następnie losujemy wartość z przedziału i na potrzeby testów wyświetlamy ją na ekranie.

```
number = random.randint(start, stop)
print("randomly selected number:", number)
```

W tym programie użyjemy pętli `while`, ponieważ nie wiemy, ile razy użytkownik będzie wprowadzał błędną wartość.

```
while True:
```

Warunek pętli ustawiliśmy na `True`, ponieważ w tym momencie nie umiemy go sprecyzować.
Następnie w każdej iteracji wczytujemy od użytkownika wartość:

```
    x = int(input("Enter number"))
```

i porównujemy ją z wylosowaną liczbą:

```
    if x==number:
        print("Great!")
        break
    elif x<number:
        print("Enter bigger number")
    else:
        print("Enter smaller number")
```

Jeżeli użytkownik zgadnie, to poza wypisaniem komunikatu wykona się również instrukcja `break`, która przerwie działanie pętli i umożliwi zakończenie programu. Jeżeli wprowadzona wartość nie zgadza się z wylosowaną, użytkownik otrzyma stosowny komunikat, czy wprowadził za mało, czy za dużo.

Pełny kod programu wyglądać może następująco:

```
import random

start = int(input("Enter start"))
stop = int(input("Enter stop"))
if start>stop:
    start, stop = stop, start

number = random.randint(start, stop)
#print("randomly selected number:", number)

while True:
    x = int(input("Enter number"))
    if x==number:
        print("Great!")
        break
    elif x<number:
        print("Enter bigger number")
    else:
        print("Enter smaller number")
```

3.7 Zadanie 7 – losowe liczby podzielne przez 3

Napisz program, który wylosuje podaną przez użytkownika liczbę podzielnych przez 3 wartości z przedziału <1, 100>.

Importujemy moduł z funkcjami losującymi:

```
import random
```

oraz wczytujemy od użytkownika, ile liczb chce wylosować:

```
counter = int(input("How may numbers do you need?"))
```

Ustawiamy zmienną pomocniczą do przechowywania ile właściwych liczb już wylosowaliśmy:

```
i = 0
```

Ponieważ nie jesteśmy w stanie określić, ile iteracji będzie potrzebnych, użyjemy pętli `while`:

```
while i<counter:
```

Wewnątrz instrukcji iteracyjnej losujemy wartość z przedziału $< 1, 100 >$:

```
    x = random.randint(1, 100)
```

i sprawdzamy, czy jest ona podzielna przez 3. Jeżeli nie, to korzystając z instrukcji `continue`, wracamy do miejsca sprawdzenia warunku, pomijając dalsze instrukcje w pętli, czyli wyświetlenie wylosowanej wartości i zwiększenie licznika.

```
    if x%3 != 0:
        continue
    print(f"random value: {x}")
    i+=1
```

Ważne, aby inkrementacja licznika `i` odbyła się tylko w przypadku, gdy wylosowana liczba spełnia kryterium podzielności przez 3.

Pełny kod programu będzie miał następującą postać:

```
import random
counter = int(input("How may numbers do you need?"))
i = 0
while i<counter:
    x = random.randint(1, 101)
    if x%3 != 0:
        continue
    print(f"random value: {x}")
    i+=1
```

3.8 Zadanie 8 – liczba pierwsza

Napisz program sprawdzający, czy wczytana od użytkownika liczba jest liczbą pierwszą.

Aby sprawdzić, czy dana liczba jest liczbą pierwszą, należy sprawdzić, czy ma ona dzielniki inne niż 1 i samą siebie. Należy pamiętać, że pierwszość jest cechą liczb nieujemnych, przy czym zarówno 0, jak i 1 nie są liczbami pierwszymi.

Program zaczynamy od pobrania wartości od użytkownika:

```
number = int(input("Enter number"))
```

i sprawdzenia, czy wprowadzona wartość kwalifikuje się do dalszego testowania:

```
if number <= 0:
    print("Invalid number")
else:
```

Jeżeli nie, przystępujemy do sprawdzania, czy wprowadzona liczba ma dzielniki. Jeśli tak, to nie jest to liczba pierwsza, a jeżeli nie, to jest to liczba pierwsza.

```
    for i in range(2, number):
        if number % i == 0:
            print(f"{number} is not a prime number")
            break
```

Zmienna sterująca `i` pętli `for` przybiera kolejne wartości z przedziału $< 2, n)$. Instrukcją warunkową sprawdzamy, czy przy dzieleniu zmiennej `number` przez `i` występuje reszta z dzielenia. W przypadku, gdy reszty nie ma, znaczy to, że `i` jest dzielnikiem `number`. Przerywamy pętlę, wypisując komunikat, gdyż wystarczy nam informacja o pierwszym dzielniku i dalsze dzielenia nie mają już sensu.

Musimy jeszcze zadbać o informację, że liczba jest liczbą pierwszą. Skąd o tym wiemy? Wykonanie wszystkich iteracji pętli `for` bez wcześniejszego zakończenia jest równoznaczne z nieznalezieniem żadnego dzielnika. Możemy w tym miejscu użyć instrukcji `else` połączonej z pętlą:

```
    else:
        print(f"{number} is a prime number")
```

Kod pełnego programu może mieć następującą postać:

```
number = int(input("Enter a number:"))
if number <= 0:
    print("Invalid number")
else:
    for i in range(2, number):
        if number % i == 0:
            print(f"{number} is not a prime number")
            break
    else:
        print(f"{number} is a prime number")
```

Powyższy kod można usprawnić, korzystając z następujących własności:

- jedyną parzystą liczbą pierwszą jest liczba 2,
- szukając kolejnych dzielników, możemy pominąć wartości parzyste, pierwszy możliwy dzielnik to liczba 3,
- zakres poszukiwania dzielników liczby można ograniczyć z góry przez $\sqrt{a}$, ponieważ jest to ostatnia liczba, która może być dzielnikiem liczby a.

Pierwszą z wymienionych własności zapiszemy następująco, wprowadzając do instrukcji warunkowej dodatkowy wariant:

```
    elif number%2 == 0:
        print(f"{number} is not a prime number")
```

Wykorzystanie dwóch następnych własności zawrzemy w sekwencji `range` pętli `for` poszukującej dzielników:

```
        for i in range(3, int(number**0.5)+1, 2):
```

Pierwsza modyfikacja dotyczy początku zakresu – tym razem zaczynamy poszukiwania od wartości 3. Zmianie uległ również koniec przedziału dla funkcji `range`.
W *Pythonie* operator `**` jest operatorem potęgowania. Jego lewy argument to podstawa potęgi, a prawy – jej wykładnik. Obydwa te parametry mogą być liczbami rzeczywistymi. Ułamkowy wykładnik jest równoznaczny z pierwiastkiem. Zatem zapis:

```
number**0.5
```

oznacza wyliczenie pierwiastka kwadratowego. A ponieważ parametrami funkcji `range` mogą być tylko wartości całkowite, musimy zaokrąglić spierwiastkowaną wartość. Zwiększenie o 1 jest konieczne, by uwzględnić pierwiastek sprawdzanej liczby wśród jej dzielników.
Krok dla funkcji `range` ustawiamy na dwa, by wyeliminować spośród potencjalnych dzielników liczby parzyste.

Kod po poprawkach będzie wyglądał następująco:

```
number = int(input("Enter number"))

if number <= 0:
    print("Invalid number")
else:
    if number == 2:
        print(f"{number} is a prime number")
    elif number%2 == 0:
        print(f"{number} is not a prime number")
    else:
        for i in range(3, int(number**0.5)+1, 2):
            if number % i == 0:
                print(f"{number} is not a prime number")
                break
        else:
            print(f"{number} is a prime number")
```

3.9 Zadanie 9 – obliczanie sumy cyfr liczby

Napisz program obliczający sumę cyfr danej liczby naturalnej.

Program rozpoczynamy wczytaniem liczby i wyzerowaniem zmiennej przechowującej sumę:

```
number = int(input("Enter a number"))
sum = 0
```

Aby rozwiązać to zadanie, potrzebne będą nam dwie operacje: *reszta z dzielenia*, dzięki której wyłuskamy z liczby wartość jej ostatniej cyfry, oraz *dzielenie całkowite*, które umożliwi nam usunięcie z liczby jej ostatniej cyfry. Te dwie operacje wykonujemy tak długo, dopóki testowana liczba jest większa od 0:

```
while number>0:
    sum += number % 10
    number = int(number/10)
```

Pierwsza instrukcja wewnątrz pętli zwiększa sumę o wartość reszty z dzielenia przez 10 – to jest ostatnia cyfra naszej liczby. Druga dzieli liczbę przez 10, dzięki czemu pozbywamy się ostatniej cyfry. W tym miejscu konieczne jest zaokrąglenie podzielonej liczby do wartości całkowitej.

Kod uzupełniony o wyświetlanie wyniku wygląda następująco:

```python
number = int(input("Enter a number"))
sum = 0
while number>0:
    sum += number % 10
    number = int(number/10)
print(f"Sum of digits of {number} equals {sum}")
```

Rozdział 4

Typ napisowy

Napisy w *Pythonie* (jak również w innych językach programowania) to ciąg znaków, które służą do przechowywania informacji (danych) tekstowych. Napisem jest każdy ciąg znaków, który ujęliśmy w cudzysłów lub apostrofy. Warunkiem jest jednak, by początek i koniec napisu były ograniczone tym samym znakiem:

```
s1 = "Hello world!"
s2 = 'Hello world'
```

Umieszczenie cudzysłowu w napisie otoczonym cudzysłowami (lub apostrofu w napisie otoczonym apostrofami) jest możliwe pod warunkiem wyłączenia specjalnego znaczenia znaku. W tym celu znak do wyłączenia należy poprzedzić znakiem \ (ang. *backslash*).

```
s3 = "And the Oscar goes to the \"Dune\"!"
s4 = 'I don\'t know, I don\'t understand'
```

4.1 Zadanie 1 – wyznaczanie długości napisu

Napisz program, który wczytuje napis i wyświetla na ekranie jego długość.

Program zaczynamy od wczytania napisu:

```
s = input("Enter a string")
```

Ponieważ interesuje nas wczytanie napisu, a nie liczb, nie używamy tu poznanych wcześniej funkcji `int()` ani `float()`.
Następnie wyznaczamy jego długość, korzystając z funkcji `len`:

```
d = len(s)
```

i wyświetlamy na ekranie komunikat:

```
print(f"Length of a string {s} equals {d}.")
```

Możemy również wywołać funkcję `len` bezpośrednio w trakcie wypisywania:

```
print(f"Length of a string {s} equals {len(s)}")
```

Kod pełnego programu może wyglądać następująco:

```
s = input("Enter a string")
print(f"Length of a string {s} equals {len(s)}")
```

4.2 Zadanie 2 – zliczanie wystąpień znaków

Napisz program, który wczytuje od użytkownika dwa napisy i sprawdza, ile razy drugi występuje w pierwszym. Program podaje również informację, na której pozycji znajduje się pierwsze i ostatnie wystąpienie.

Wczytujemy dwa napisy:

```
s1 = input("Enter the first string")
s2 = input("Enter the second string")
```

Funkcja `count` zwraca liczbę wystąpień jednego łańcucha w drugim:

```
how_many = s1.count(s2)
print(f"String \'{s2}\' appears in \'{s1}\' {how_many} times.")
```

Zwróć uwagę na sposób wywołania. Ponieważ szukamy wystąpień `s2` wewnątrz `s1`, zapisujemy je następująco: `s1.count(s2)`. Wyświetlamy wynik na ekranie. Dalsza część zadania ma sens tylko wtedy, gdy `s2` występuje w `s1`. Dlatego używamy instrukcji warunkowej:

```
if how_many > 0:
    first = s1.find(s2);
    last = s1.rfind(s2)
```

Korzystając z funkcji `find` oraz `rfind`, znajdujemy indeksy, pod którymi zaczyna się kolejno pierwsze i ostatnie wystąpienie. Można powiedzieć, że funkcja `rfind` wyszukuje pierwsze od końca wystąpienie.

Cały program, uzupełniony o wypisanie na ekranie pozycji wystąpień, wygląda następująco:

```
s1 = input("Enter the first string")
s2 = input("Enter the second string")
how_many = s1.count(s2)
print(f"String \'{s2}\' appears in \'{s1}\' {count} times.")

if how_many > 0:
    first = s1.find(s2);
    last = s1.rfind(s2)
    print(f"An index of the first occurrence: {first}.")
    print(f"An index of the last occurrence: {last}.")
```

4.3 Zadanie 3 – wyświetlanie podłańcuchów

Napisz program, który z ciągu znaków podanego przez użytkownika wypisze na ekranie jego pierwszy znak, dwa pierwsze znaki i wszystkie znaki poza pierwszymi dwoma.
Wypisze także ostatni znak, dwa ostatnie znaki oraz wszystkie znaki napisu poza ostatnimi dwoma.
Jeśli ciąg jest krótszy niż 2 znaki, wypisze na ekranie stosowny komunikat.
Następnie wyświetli co drugi znak napisu od początku i co drugi od końca.

Zaczynamy od wczytania napisu od użytkownika:

```
s = input("Enter a string: ")
```

Napisy są sekwencjami znaków. Każdy typ sekwencyjny pozwala na dostęp do każdego swojego elementu z osobna. Aby uzyskać dostęp do znaku na określonej pozycji, podajemy jej indeks (numer porządkowy liczony od 0) w nawiasach kwadratowych bezpośrednio po nazwie zmiennej napisowej.
Poniższy kod wyświetla zatem pierwszy znak napisu:

```
print(f"The first character of {s} is {s[0]}")
```

Pierwsze dwa znaki napisu otrzymamy, podając zakres indeksów:

```
print(f"The firsts two characters of {s} are {s[0:2]}")
```

Jeżeli początek zakresu wynosi 0, to możemy go pominąć:

```
print(f"The firsts two characters of {s} are {s[:2]}")
```

Napis bez pierwszych dwóch znaków to napis zaczynający się od indeksu 2:

```
print(f"String {s} without two firsts characters: {s[3:]}")
```

Koniec zakresu możemy w tym wypadku pominąć – jest to tożsame z zapisem `len(s)`.

Ostatni znak napisu moglibyśmy uzyskać, używając funkcji wyznaczającej długość napisu:

```
print(f"The last character of {s} is {s[len(s)-1]}")
```

Ale możemy również skorzystać z tego, że *Python* umożliwia podawanie ujemnych indeksów:

```
print(f"The last character of {s} is {s[-1]}")
```

Indeks -1 oznacza pierwszy znak od końca.

Analogicznie wyświetlimy dwa ostatnie znaki – podając przedział od -2 do końca napisu:

```
    print(f"The lasts two characters of {s} are {s[-2:len(s)]}")
```

Ponownie, koniec przedziału możemy pominąć, upraszczając zapis do:

```
    print(f"The lasts two characters of {s} are {s[-2:]}")
```

Napis bez dwóch ostatnich znaków wyświetlimy za pomocą przedziału od 0 do drugiego od końca:

```
    print(f"String {s} without two last characters: {s[0:-2]}")
```

Tym razem również możemy uprościć zapis, pomijając początek zakresu:

```
    print(f"String {s} without two last characters: {s[:-2]}")
```

Powyższe instrukcje można wykonać pod warunkiem, że napis jest dłuższy niż 2 znaki:

```
if (len(s)>=2):
    print(f"The firsts two characters of {s} are {s[:2]}")
    print(f"String {s} without two firsts characters: {s[2:]}")

    print(f"The lasts two characters of {s} are {s[-2:]}")
    print(f"String {s} without two last characters: {s[:-2]}")
else:
    print(f"String {s1} is to short")
```

Podobnie jak w było przypadku pętli `for`, przy odwołaniu do typów sekwencyjnych możemy określić nie tylko początek i koniec przedziału, lecz także krok. Wyświetlenie co drugiego znaku oznacza krok równy 2:

```
print(f"Every second character from the beginning: {s[::2]}")
```

Pomijamy krańce przedziału, które zostaną domyślnie ustawione na 0 oraz $len(s) - 1$.

Jeżeli chcemy wyświetlić napis od końca, to krok musi mieć wartość ujemną, a początek zakresu ustawiamy na -1:

```
print(f"Every second character from the end: {s[-1::-2]}")
```

4.4 Zadanie 4 – sprawdzanie, czy napis jest palindromem

Napisz program, który wczyta od użytkownika ciąg znaków. Sprawdź, czy ciąg ten jest palindromem, np. kot – nie, kajak – tak.

Wczytujemy napis:

```
s = input("Enter a string")
```

Dany wyraz jest palindromem, jeżeli jego pierwsza litera jest taka sama jak ostatnia, druga jest taka sama jak przedostatnia, trzecia jest taka sama jak trzecia od końca itd., dopóki nie znajdziemy się w połowie wyrazu. Przy czym warto zauważyć, że dla słowa o parzystej liczbie liter wykonamy tyle samo porównań, ile dla wyrazu o nieparzystej liczbie liter. Zatem iterując po napisie od indeksu zerowego do indeksu $len(s)/2$, porównujemy ze sobą dwa elementy z początku i końca napisu:

```
for i in range(int(len(s)/2)):
    if s[i]!=s[len(s)-1-i]:
        print(f"{s} is not a palindrome")
        break
```

Jeżeli znajdziemy pierwszą niezgodność, przerywamy pętlę, wypisując komunikat.
Instrukcja `else` do pętli `for` wykona się tylko wtedy, gdy pętla nie zostanie przerwana. W takim wypadku wyświetlamy komunikat, że napis jest palindromem:

```
else:
    print(f"{s} is not a palindrome")
```

Cały kod programu prezentuje się następująco:

```
s = input("Enter a string")

for i in range(int(len(s)/2)):
    if s[i]!=s[len(s)-1-i]:
        print(f"{s} is not a palindrome")
        break
else:
    print(f"{s} is a palindrome")
```

Te zadanie można również wykonać inaczej. Napis jest palindromem, jeżeli czytany od początku jest taki sam jak czytany od końca. Wykorzystując możliwości, jakie oferuje indeksowanie napisów w *Pythonie*, napis *od końca* definiujemy jako:

```
s[::-1]
```

Zatem program może wyglądać jak poniżej:

```
s = input("Enter a string: ")
if s==s[::-1]:
    print(f"{s} is a palindrome")
```

```
else:
    print(f"{s} is not a palindrome")
```

Palindromami są nie tylko pojedyncze wyrazy. Zdanie *Kobyła ma mały bok* jest również palindromem. Po uruchomieniu programu okaże się jednak, że napisany kod nie działa prawidłowo. Obecna wersja porównuje po prostu kolejne znaki napisu, bez znaczenia jakie są. Wielkość liter ma również znaczenie.
Aby rozwiązać ten problem, możemy pozbyć się z napisu białych znaków oraz ujednolicić wielkość liter.

```
s = s.replace(" ", "")
s = s.lower()
```

Metoda `.replace` zamienia wystąpienia pierwszego parametru drugim, a `.lower` zamienia wszystkie litery na małe. Obydwie metody zwracają nowy napis, który przypisujemy naszej zmiennej `s`.

4.5 Zadanie 5 – obliczanie sumy cyfr liczby

Napisz program obliczający sumę cyfr danej liczby naturalnej.

Sumę cyfr liczby wyznaczaliśmy już w Przykładzie 3.9. Tym razem rozwiążemy zadanie, wykorzystując napisy, a nie liczby całkowite.
Wczytując wartość od użytkownika, nie zamieniamy jej na liczbę:

```
number = input("Enter a number: ")
```

Łańcuch `number` przechowuje wczytaną wartość.
Iterując po każdej cyfrze napisu, konwertujemy ją na liczbę (korzystając z `int`) i dodajemy do zmiennej `sum`:

```
sum = 0
for digit in number:
    sum += int(digit)
print(f"Sum of digits of {number} equals {sum}")
```

Powyższy kod będzie działał tylko w przypadku, gdy użytkownik wprowadzi napis składający się z samych cyfr. W przeciwnym wypadku wykonanie zakończone zostanie błędem: *ValueError: invalid literal for int() with base 10.*
Aby temu zapobiec, musimy sprawdzić, czy wczytany napis zawiera tylko cyfry. W tym celu wykorzystamy metodę `.isdigit()`:

```
if number.isdigit():
    sum = 0
    for digit in number:
```

```
        sum += int(digit)
    print(f"Sum of digits of {number} equals {sum}")
else:
    print("This is not a number!")
```

Zwraca ona `True`, jeżeli wszystkie znaki w napisie są cyframi. Możemy ją stosować wymiennie z funkcją `.isnumeric()`. Podobnymi funkcjami są `.isapha()`, która zwraca wynik sprawdzenia, czy wszystkie znaki napisu są literami i napis składa się przynajmniej z jednego znaku, oraz `.isalnum()`, sprawdzająca czy wszystkie znaki napisu są znakami alfanumerycznymi i napis składa się przynajmniej z jednego znaku.

4.6 Zadanie 6 – generator haseł

Napisz program, który wygeneruje losowe hasło o podanej przez użytkownika długości. Hasło tworzone jest z małych i wielkich liter, cyfr oraz znaków interpunkcyjnych.

Zaczynamy od pobrania długości hasła i stworzenia pustego napisu:

```
length = int(input("Enter length of your password: "))
password = ""
```

Następnie w pętli losujemy wartość od 1 do 4, bo tyle mamy możliwych zbiorów znaków.

```
for i in range(length):
    character = ""
    choice = random.randint(1, 4)
```

Korzystanie z funkcji `random.randint` wymaga oczywiście zaimportowania pakietu `random`.
W zależności od wylosowanej wartości `choice` losujemy znak.
Python w pakiecie `string` udostępnia zdefiniowane napisy zawierające małe litery – `string.ascii_lowercase`, wielkie litery – `string.ascii_uppercase`, cyfry – `string.digits` oraz znaki przestankowe – `string.punctuation`.

```
    if choice == 1:
        character = string.ascii_lowercase[random.randint(0,
                    len(string.ascii_lowercase))-1]
```

Do zmiennej `character` przypisywany jest znak z napisu `string.ascii_lowercase` o indeksie wylosowanym z przedziału 0 – *liczba_liter*. W analogiczny sposób możemy wylosować wielkie litery, cyfry oraz znaki przestankowe.

Powyższy zapis jest długi i łatwo w nim o pomyłkę. Zamiast definiować krańce przedziału dla funkcji `random.randint`, możemy skorzystać z funkcji `random.choice`, która zwraca losowo wybrany element niepustej sekwencji przekazanej w parametrze, w tym wypadku w napisie `string.ascii_lowercase`:

```
    if choice == 1:
        character = random.choice(string.ascii_lowercase)
```

Łącząc powyższe informacje, otrzymamy poniższy kod:

```
import random
import string

length = int(input("Enter length of your password: "))
password = ""
for i in range(length):
    character = ""
    choice = random.randint(1, 4)
    if choice == 1:
        character = random.choice(string.ascii_lowercase)
    elif choice == 2:
        character = random.choice(string.ascii_uppercase)
    elif choice == 3:
        character = random.choice(string.digits)
    else:
        character = random.choice(string.punctuation)
    password += character
print(password)
```

4.7 Zadanie 7 – anonimizacja danych

Napisz program, który wczyta od użytkownika jego nazwisko i wypisze je w zanonimizowanej postaci. Anonimizacja polega na zostawieniu pierwszej litery i trzech ostatnich liter w oryginalnej formie oraz zastąpienie wszystkich środkowych przez znak '' ("Kowalski" => "K****ski"). Jeżeli nazwisko jest krótsze niż trzy litery, wszystkie zostają zastąpione znakiem '*'.*

Zaczynamy od wczytania nazwiska od użytkownika i stworzenia pustego napisu, który będzie przechowywał zanonimizowaną postać:

```
name = input("Enter your name: ")
coded = ""
```

Jeżeli nazwisko jest krótsze niż trzy znaki, wszystkie litery zostają zastąpione znakiem '*'.

```
if len(name)<=3:
    coded = "*"*len(name)
```

Python umożliwia powielanie napisów. Używamy w tym celu operatora mnożenia `'*'`. Zauważ jednak, że nie jest to typowe mnożenie, jak w przypadku liczb, tylko powielanie (powtórzenie). Istotne również jest, że liczba powieleń musi być dodatnią liczbą całkowitą typu `int`.
Powyższy kod przypisze zatem do napisu `coded` tyle gwiazdek `'*'`, ile wynosi długość nazwiska.

Jeżeli napis jest dłuższy, to jako pierwszy znak zakodowanego nazwiska wstawiamy pierwszy znak nazwiska `name[0]`. Następnie umieszczamy $len(name) - 4$ symbole gwiazdek. Na koniec dodajemy jeszcze trzy ostatnie znaki nazwiska `name[-3::]`:

```
    coded += name[0]
    coded += "*"*(len(name)-4)
    coded += name[-3::]
```

Kompletny kod programu będzie wyglądał jak poniżej:

```
name = input("Enter your name: ")
coded = ""

if len(name)<=3:
    coded = "*"*len(name)
else:
    coded += name[0]
    coded += "*"*(len(name)-4)
    coded += name[-3::]
print(f"Coded name: {coded}")
```

Rozdział 5

Listy

Lista (ang. *list*) w *Pythonie* jest typem sekwencyjnym o następujących własnościach:

- Elementami listy mogą być obiekty dowolnego typu, pojedyncza lista także może zawierać elementy różnych typów.
- Początek i koniec listy oznaczamy nawiasami kwadratowymi.
- Elementy listy rozdzielamy między sobą przecinkami.
- Można dowolnie zmieniać zawartość istniejącej listy.

Przetestuj poniższe instrukcje:

```
list1 = []
list2 = [2]
list3 = [1, 2, 3, 4, 5]
list4 = ["alice", "has", "cats"]
list5 = ["howard", "has", 5, "dogs"]
list6 = list4+list5

list7 = list4
list7 += list5

print(list1)
print(list2)
print(list3)
print(list4)
print(list5)
print(list6)
print(list7)
```

5.1 Zadanie 1 – zliczanie wartości parzystych

Napisz program, który wczyta od użytkownika 10 liczb i umieści je na liście. Następnie wyświetl wprowadzone wartości na ekranie oraz policz, ile jej elementów jest liczbami parzystymi.

Program zaczynamy od stworzenia pustej listy:

```
list = []
```

Następnie w pętli wczytujemy wartości od użytkownika i korzystając z metody `.append()`, umieszczamy je kolejno na liście:

```
for i in range(10):
    x = int(input("Enter a number: "))
    list.append(x)
```

Wypełnioną listę możemy wyświetlić element po elemencie, ale również możemy ją przekazać jako parametr do funkcji `print()`:

```
print(list)
```

Przechodzimy do zliczenia elementów parzystych. Zmienna `counter` będzie przechowywała liczność tych wartości – przed rozpoczęciem zliczania musimy ją wyzerować:

```
counter = 0
```

Następnie konstruujemy pętlę iterującą po kolejnych elementach listy:

```
for x in list:
```

Zmienna `x` będzie przyjmowała elementy z listy `list`. Wewnątrz pętli sprawdzamy, czy aktualna wartość przechowywana w `x` jest parzysta i jeśli tak, to zwiększamy licznik:

```
    if x%2 == 0:
        counter += 1
```

Po sprawdzeniu wszystkich elementów listy wyświetlamy na ekranie stosowne komunikaty:

```
if counter == 0:
    print("There is no even numbers.")
elif counter == 1:
    print("There is 1 even number.")
else:
    print(f"There are {counter} even numbers.")
```

Pełny kod programu wygląda zatem następująco:

```
print("Enter 10 numbers: ")
list = []
for i in range(10):
    x = int(input("Enter number"))
    list.append(x)
print(list)
counter = 0
```

```
for x in list:
    if x%2 == 0:
        counter += 1
if counter == 0:
    print("There is no even numbers.")
elif counter == 1:
    print("There is 1 even number.")
else:
    print(f"There are {counter} even numbers.")
```

5.2 Zadanie 2 – obliczanie iloczynu elementów listy

Napisz program, który wczyta od użytkownika 10 liczb, umieści je na liście. Następnie wyświetl wprowadzone wartości na ekranie i wyznacz iloczyn elementów na liście.

Deklarujemy pustą listę oraz wypełniamy ją wartościami wczytanymi z klawiatury:

```
list = []
for i in range(10):
    x = int(input("Enter a number: "))
    list.append(x)
```

Następnie tworzymy zmienną `product`, w której znajdzie się iloczyn elementów listy. Nadajemy jej wartość 1 – wartość neutralną dla mnożenia:

```
product = 1
```

Wyznaczamy iloczyn:

```
for x in list:
    produkt *= x
```

i wypisujemy wynik na ekranie:

```
print(f"Product equals = {product}")
```

Pełny kod programu ma zatem następującą postać:

```
print("Enter 10 numbers: ")
list = []
for i in range(10):
    x = int(input("Enter a number: "))
    list.append(x)
```

```
product = 1
for x in list:
    product *= x
print(f"Product equals = {product}")
```

5.3 Zadanie 3 – obliczanie średniej nieparzystych elementów

Napisz program, który utworzy i wypisze listę złożoną z n losowych liczb z przedziału <1, 50>. Liczbę elementów listy `n` *pobierz od użytkownika. Następnie program obliczy średnią arytmetyczną jej nieparzystych elementów.*

Ponieważ w tym zadaniu elementy listy są losowane, zaczynamy od zaimportowania biblioteki `random`:

```
import random
```

Następnie pobieramy od użytkownika rozmiar listy:

```
n = int(input("Enter number of elements: "))
```

Tworzymy pustą listę i wypełniamy ją losowymi wartościami:

```
list = []
for i in range(n):
    list.append(random.randint(1, 50))
```

Aby obliczyć średnią elementów nieparzystych, tworzymy dwie zmienne: `avg`, której użyjemy początkowo do sumowania wartości, oraz `counter`, która będzie przechowywała liczbę elementów nieparzystych na liście.

```
avg = 0
counter = 0
```

Przechodzimy listę element po elemencie i sprawdzamy, czy bieżąca wartość przechowywana w zmiennej `x` jest nieparzysta. Jeżeli tak, aktualizujemy zmienne `avg` oraz `counter`:

```
for x in list:
    if x%2 == 1:
        counter += 1
        avg += x
```

Może się zdarzyć, że żadna z wylosowanych liczb na liście nie jest nieparzysta. W takim wypadku nie możemy obliczyć średniej. Sprawdzimy to, przyrównując zmienną `counter` do zera:

```
if counter == 0:
    print("There are no odd numbers")
else:
    avg /= counter
    print(f"Average of odd numbers equals {avg}")
```

Pełny kod programu wygląda jak poniżej:

```
import random
n = int(input("Enter number of elements: "))
list = []
for i in range(n):
    list.append(random.randint(1, 50))
print(list)
avg = 0
counter = 0
for x in list:
    if x%2 == 1:
        counter += 1
        avg += x
if counter == 0:
    print("There are no odd numbers")
else:
    avg /= counter
    print(f"Average of odd numbers equals {avg}")
```

5.4 Zadanie 4 – zmiana wartości elementów na liście

Napisz program, który utworzy i wypisze listę złożoną z n losowych liczb z przedziału <-50, 50>. Rozmiar listy **n** *podaje użytkownik. Następnie program zmieni wartość każdego elementu na przeciwny i wyświetli zmodyfikowaną listę.*

Listę tworzymy analogicznie jak we wcześniejszym zadaniu:

```
import random
n = int(input("Enter number of elements: "))
list = []
for i in range(n):
    list.append(random.randint(-50, 50))
```

Ponieważ chcemy zmodyfikować wartość elementu, a nie tylko odczytać jego wartość, musimy zmienić sposób iteracji po liście. Tym razem zmienną sterującą pętli będzie indeks elementu przechowywany w zmiennej `i`:

```
for i in range(len(list)):
```

Funkcja `len()` zwraca nam rozmiar listy.
Odwołanie do elementu realizujemy poprzez operator indeksowania, czyli `[]`:

```
    list[i] = -list[i]
```

Najpierw po prawej stronie operatora pobieramy i mnożymy przez -1 wartość i-tego elementu, a następnie przypisujemy nową wartość w to samo miejsce na liście.

Kod programu uzupełniony o wyświetlanie listy przed zmianą i po zmianie wygląda zatem tak:

```
import random
n = int(input("Enter number of elements: "))
list = []
for i in range(n):
    list.append(random.randint(-50, 50))
print(list)
for i in range(len(list)):
    list[i] = -list[i]
print(list)
```

5.5 Zadanie 5 – usuwanie wartości

Napisz program, który utworzy i wypisze listę złożoną z n losowych liczb z przedziału <1, 50>. Rozmiar listy **n** *podaje użytkownik. Następnie program usunie z listy elementy należące do przedziału <11, 20> i wyświetli zmodyfikowaną listę.*

Listę tworzymy podobnie jak we wcześniejszych zadaniach:

```
import random
n = int(input("Enter number of elements: "))
list = []
for i in range(n):
    list.append(random.randint(10, 22))
```

Następnie dla każdej liczby z przedziału $< 11, 20 >$

```
for x in range(11, 21):
```

sprawdzamy, czy występuje ona na liście:

```
    if x in list:
```

Jeżeli dana wartość występuje na liście, to wynikiem użytego operatora `in` jest `True` i wykona się poniższa instrukcja:

```
        list.remove(x)
```

Metoda `.remove()` służy do usunięcia z listy wartości przekazanej w parametrze. Jednakże po wyświetleniu listy na ekranie okaże się, że nadal znajdują się na niej wartości z przedziału $< 11, 20 >$. Dlaczego? Otóż `.remove()` usuwa tylko pierwsze wystąpienie wartości na liście. Jeżeli dana wartość została wylosowana więcej niż raz, musimy wywołać metodę `.remove()` kilkukrotnie.
W tym celu musimy najpierw sprawdzić, ile razy każda wartość z przedziału pojawiła się na liście. Służy do tego metoda `.count()`:

```
    c = list.count(x)
```

Następnie w pętli c razy wywołamy metodę `.remove()`

```
    for i in range(c):
        list.remove(x)
```

Tym razem z listy zostaną usunięte wszystkie wystąpienia każdej liczby z przedziału $< 11, 20 >$:

```
import random
n = int(input("Enter number of elements: "))
list = []
for i in range(n):
    list.append(random.randint(10, 22))
print(list)
for x in range(11, 21):
    c = list.count(x)
    for i in range(c):
        list.remove(x)
print(list)
```

5.6 Zadanie 6 – wyszukiwanie powtarzających się nazwisk

Napisz program, który pobierze od użytkownika imiona i nazwiska uczestników dwóch szkoleń (każde szkolenie ma oddzielną listę obecności). Następnie zrób listę osób, które wzięły udział w obydwu szkoleniach, oraz listę wszystkich uczestników, którzy wzięli udział przynajmniej w jednym ze szkoleń. Stworzone listy posortuj alfabetycznie.

Program zaczynamy od stworzenia pustych list:

```
names1 = []
names2 = []
```

Następnie pobieramy liczbę uczestników pierwszego szkolenia i wczytujemy nazwiska, umieszczając je na pierwszej liście:

```
n = int(input("Enter number of participants of first training: "))
print("Enter the participants")
for i in range(n):
    names1.append(input())
```

W analogiczny sposób wypełniamy drugą listę:

```
n = int(input("Enter number of participants of second training: "))
print("Enter the participants")
for i in range(n):
    names2.append(input())
```

Użyliśmy tej samej zmiennej `n` do pobrania rozmiaru drugiej listy. Możemy tak zrobić, ponieważ liczbę elementów na liście możemy pobrać w każdej chwili funkcją `len()` i nie musimy jej trzymać w dodatkowej zmiennej.
Następnie tworzymy pustą listę na przechowywanie uczestników obydwu szkoleń:

```
duplicates = []
```

Idąc po liście pierwszej, sprawdzamy operatorem `in`, czy dana osoba występuje również na liście drugiej. Jeżeli tak, umieszczamy ją na liście *"duplikatów"*:

```
for name in names1:
    if name in names2:
        duplicates.append(name)
```

Musimy jeszcze posortować listę alfabetycznie. Służy do tego metoda `.sort()`.

```
duplicates.sort()
```

Domyślnym kierunkiem sortowania jest porządek rosnący, w tym wypadku alfabetyczny. Gdybyśmy chcieli ułożyć elementy malejąco, to metodę `sort()` wywołujemy z parametrem `reverse`:

```
duplicates.sort(reverse=True)
```

Listę osób biorących udział przynajmniej w jednym szkoleniu zaczynamy od wypełnienia jej osobami z pierwszej listy:

```
uniques = names1.copy()
```

Metoda `.copy()`tworzy kopię listy `names1` i przypisuje ją do listy `uniques`. Dlaczego nie użyliśmy po prostu przypisania: `uniques = names1`? Użycie

`.copy()` gwarantuje nam, że obydwie listy będą od siebie niezależne i modyfikacja elementów jednej nie wpłynie na elementy drugiej. Z kolei operator przypisania tworzy po prostu drugą nazwę dla istniejącej już listy i tak naprawdę mamy tylko jedną pod dwiema nazwami. Jakakolwiek zmiana widoczna jest bez względu na to, której nazwy listy użyjemy.
Następnie, iterując po drugiej liście `names2`, sprawdzamy, czy jej bieżący element już wystąpił w `uniques`. Jeżeli nie, to go wstawiamy:

```
for name in names2:
    if name not in uniques:
        uniques.append(name)
```

Pełny kod programu uzupełniony o sortowanie i wyświetlanie nowych list przedstawia się następująco:

```
names1 = []
names2 = []
n = int(input("Enter number of participants of first training: "))
print("Enter the participants")
for i in range(n):
    names1.append(input())

n = int(input("Enter number of participants of second training: "))
print("Enter the participants")
for i in range(n):
    names2.append(input())

duplicates = []
uniques = []

for name in names1:
    if name in names2:
        duplicates.append(name)
duplicates.sort()
print(duplicates)

uniques = names1.copy()
for name in names2:
    if name not in uniques:
        uniques.append(name)
uniques.sort()
print(uniques)
```

5.7 Zadanie 7 – znajdowanie czynników pierwszych liczby

Napisz program tworzący listę, której elementami są czynniki pierwsze podanej przez użytkownika liczby.
Przykład:
Podana liczba: 24
*Czynniki: 24 = 2*2*2*3*
Podana liczba: 392
*Czynniki: 392 = 2*2*2*7*7*

Czynnikiem pierwszym jest dowolna liczba pierwsza, która dzieli bez reszty daną liczbę naturalną złożoną.
Zadanie podzielimy na dwa etapy. W pierwszym znajdziemy potencjalne dzielniki pierwsze, czyli po prostu wszystkie liczby pierwsze mniejsze od podanej liczby. Drugi etap będzie polegał na wybraniu właściwych czynników pierwszych.
Po wczytaniu od użytkownika liczby zapamiętujemy jej wartość w dodatkowej zmiennej. Przyda nam się przy wyświetlaniu końcowego komunikatu:

```
number = int(input("Enter a number: "))
orig = number
```

Tworzymy pustą listę potencjalnych dzielników:

```
factors=[]
```

Następnie w pętli wybieramy kolejne liczby z przedziału $< 2, number)$:

```
for n in range(2, number):
```

i sprawdzamy, czy jest liczbą pierwszą:

```
    for x in range (2, n):
        if n%x == 0:
            break
    else:
        factors.append(n)
```

Dla każdej liczby `x` mniejszej od `n` sprawdzamy resztę z dzielenia. Jeżeli `n` dzieli się przez `x`, przerywamy pętlę wewnętrzną, gdyż `n` nie jest liczbą pierwszą. Jeżeli pętla wewnętrzna nie zostanie przerwana, wartość `n` zostanie dopisana do listy potencjalnych dzielników `factors` – instrukcja `else` wykona się tylko wtedy, gdy połączona z nią pętla `for` wykona się do końca.
Mając wyznaczone liczby pierwsze, możemy sprawdzić, które z nich mogą być czynnikami pierwszymi naszej liczby. Zaczynamy od stworzenia pustej listy czynników:

```
prime_factors = []
```

Algorytm poszukiwania wygląda następująco. Sprawdzamy, czy liczba dzieli się przez 2. Jeżeli tak, to 2 dopisujemy do listy czynników, a samą liczbę dzielimy przez 2. Czynność tę powtarzamy, aż liczba przestanie być podzielna przez 2. Następnie powtarzamy tę procedurę, badając podzielność przez 3, 5 itd., aż rozważana liczba stanie się równa 1.
Zatem dla każdej wyznaczonej liczby pierwszej:

```
for factor in factors:
```

wykonujemy:

```
        prime_factors.append(factor)
        number = number // factor
```

Użyty operator `//` oznacza dzielenie całkowite.
Powyższe dwie instrukcje wykonujemy do momentu, gdy nasza liczba nie dzieli się przez daną liczbę pierwszą:

```
    while int(number%factor) == 0:
        prime_factors.append(factor)
        number = number // factor
```

Konieczne jest użycie funkcji `int()`, ponieważ analizujemy wartości całkowite.

Po wypełnieniu listy czynnikami wyświetlamy na ekranie komunikat. Wartość od użytkownika wczytaliśmy do zmiennej `number`, jednak modyfikowaliśmy ją, szukając czynników pierwszych. Do wypisania użyjemy zatem stworzonej wcześniej kopii `orig`.

```
print(f"{orig} =", end=' ')
for i in range(len(prime_factors)-1):
    print(f"{prime_factors[i]}*", end='')
print(f"{prime_factors[-1]}")
```

Zwróć uwagę, że pętla wykonuje się tylko do przedostatniego elementu listy czynników `range(len(prime_factors)-1)`. Dzieje się tak dlatego, że znak mnożenia wyświetlany między czynnikami nie powinien pojawić się po ostatnim elemencie. Ostatni czynnik wypisujemy po zakończeniu pętli, tym razem bez symbolu `'*'`.

Pełny kod programu wygląda następująco:

```
number = int(input("Enter a number: "))
orig = number
factors=[]
for n in range(2, number):
```

```
    for x in range (2, n):
        if n%x == 0:
            break
    else:
        factors.append(n)
prime_factors = []
for factor in factors:
    while int(number%factor) == 0:
        number = number // factor
        prime_factors.append(factor)
print(f"{orig} =", end=' ')
for i in range(len(prime_factors)-1):
    print(f"{prime_factors[i]}*", end='')
print(f"{prime_factors[-1]}")
```

Rozdział 6

Krotki

Krotka (ang. *tuple*) to kolejny po napisach i listach typ sekwencyjny *Pythona*. Wartości zapisane w krotce mogą być dowolnego typu i są indeksowane liczbami całkowitymi. W przeciwieństwie do list krotki są *niezmienne* (ang. *immutable*), stąd też używamy ich, gdy mamy do czynienia z elementami, które nie powinny być podatne na modyfikacje, jak np. spis dni tygodnia.
Krotki przydają się, gdy chcemy przechowywać listę identycznych obiektów z różnymi wartościami. Przykładowo, możemy zbudować listę o nazwie `people`, gdzie każdy element jest krotką trzech elementów: `first name`, `last name` i `zip code`. Kod pocztowy dla pierwszego elementu uzyskamy odwołaniem `people[0][2]`.

6.1 Zadanie 1 – wyznaczanie obiektów w zasięgu radaru

Napisz program, który utworzy i wypełni losowymi danymi listę n punktów (każdy punkt reprezentuje współrzędne pewnego obiektu). Następnie wypisze jej zawartość na ekranie i wczyta od użytkownika współrzędne oraz zasięg radaru. Stworzy listę tych punktów, które znajdują się w zasięgu radaru.

Zaczynamy od wylosowania listy punktów:

```
n = int(input("Enter number of points: "))
points = []
for i in range(n):
    point = ( random.randint(-20, 20), random.randint(-20, 20) )
    points.append(point)
```

Lista `points` będzie przechowywała `n` punktów, których współrzędne punktów wylosujemy z przedziału $< -20, 20 >$. Konieczne jest również zaimportowanie biblioteki `random`, z której pochodzi funkcja `random.randint()`.
Zwróć uwagę na zapis:

```
point = (random.randint(-20, 20), random.randint(-20, 20))
```

Użycie nawiasów okrągłych `()` scala nam dwie wylosowane wartości w jedną strukturę: krotkę.
Następnie wczytujemy współrzędne i zasięg radaru:

```
x = int(input("Specify the x-coordinate of the radar: "))
y = int(input("Specify the y-coordinate of the radar: "))
r = int(input("Specify the radar range: "))
```

i tworzymy z tych wartości krotkę:

```
radar = (x, y, r)
```

Deklarujemy pustą listę na punkty w zasięgu radaru:

```
within = []
```

i iterując po liście punktów, sprawdzamy, który z nich powinien się na niej znaleźć:

```
for p in points:
    if math.sqrt((p[0]-radar[0])**2+(p[1]-radar[1])**2)<=radar[2]:
        within.append(p)
```

Punkt `p` leży w zasięgu radaru, jeżeli odległość między nim a pozycją radaru jest mniejsza bądź równa zasięgowi radaru. Zwróć uwagę, w jaki sposób odwołujemy się do elementów krotek. Podobnie jak w przypadku list używamy do tego operatora indeksowania `[]`.
Funkcja `math.sqrt()` oblicza pierwiastek kwadratowy – wymagane jest zaimportowanie biblioteki `math`.

Pełny kod wraz z wyświetlaniem wartości na ekranie prezentuje się następująco:

```
import random
import math

n = int(input("Enter number of points: "))
points = []
for i in range(n):
    point = (random.randint(-20, 20), random.randint(-20, 20))
    points.append(point)
print(points)

x = int(input("Specify the x-coordinate of the radar: "))
y = int(input("Specify the y-coordinate of the radar: "))
r = int(input("Specify the radar range: "))
radar = (x, y, r)
print(f"Radar: {radar[0]} {radar[1]} {radar[2]}")

within = []
for p in points:
```

```
    if math.sqrt((p[0]-radar[0])**2+(p[1]-radar[1])**2 )<=radar[2]:
        within.append(p)
print(within)
```

6.2 Zadanie 2 – wyszukiwanie najwyższego drzewa

Dana jest lista zawierająca następujące informacje o drzewie: gatunek, wysokość, średnica pnia. Napisz program, który wypisze nazwę gatunku najwyższego drzewa. Jeżeli drzew o maksymalnej wysokości jest więcej niż jedno, wypisz dane pierwszego z listy.

Tym razem listę tworzymy na etapie pisania kodu:

```
trees = [("sosna", 2.01, 35.00), ("brzoza", 1.98, 29.50),
        ("klon", 2.50, 31.02), ("sosna", 2.12, 37.50),
        ("modrzew", 1.45, 19.56), ("dąb", 2.35, 50.00),
        ("świerk", 2.24, 42.01), ("klon", 1.53, 25.25)]
```

Tworzymy zmienną, w której będziemy przechowywać dane najwyższego drzewa, i przypisujemy jej wartość początkową na pierwszy element listy:

```
highest = trees[0]
```

Następnie w pętli przechodzimy po liście drzew, szukając najwyższego:

```
for t in trees:
    if t[1]>highest[1]:
        highest = t
```

Kod programu w całości prezentuje się następująco:

```
trees = [("sosna", 2.01, 35.00), ("brzoza", 1.98, 29.50),
        ("klon", 2.50, 31.02), ("sosna", 2.12, 37.50),
        ("modrzew", 1.45, 19.56), ("dąb", 2.35, 50.00),
        ("świerk", 2.24, 42.01), ("klon", 1.53, 25.25)]
print(trees)

highest = trees[0]
for t in trees:
    if t[1]>highest[1]:
        highest = t

print(highest)
```

6.3 Zadanie 3 – wyszukiwanie najwyższego drzewa raz jeszcze

Dana jest lista zawierająca następujące informacje o drzewie: gatunek, wysokość, średnica pnia. Napisz program, który wypisze nazwę gatunku najwyższego drzewa.

Jak wiesz, do poszczególnych elementów krotek odwołujemy się poprzez nawiasy kwadratowe `[]`. I tak właśnie robiliśmy w poprzednich zadaniach. Wyobraź sobie jednak, że informacje o drzewie to nie tylko gatunek, wysokość i średnica pnia zmierzona na jednym poziomie, lecz także jego współrzędne, średnice pnia na kilku wysokościach, historia chorób, a być może i inne dane. W takim wypadku trudniej zapamiętać, który indeks oznacza które pole.
Python dostarcza eleganckie rozwiązanie tego problemu, tzw. *nazwane krotki* (ang. *named tuples*). Są one nadal niezmienne, nadal możemy indeksować elementy za pomocą liczb całkowitych i iterować po elementach. Dodatkowo jednak możemy uzyskać dostęp do konkretnych pól przez `.nazwa_pola`.
Aby utworzyć nazwaną krotkę, musmy zaimportować `namedtuple` z modułu `collections`.

```
from collections import namedtuple
```

Jak widzisz, możemy importować nie tylko całe moduły, lecz także wybrane ich elementy.
Następnie określamy, jak nasza krotka ma wyglądać:

```
tree = namedtuple("tree", "species height diameter")
```

Pierwszy parametr funkcji `namedtuple` określa nazwę naszego typu krotki, a kolejny przyjmuje listę jej pól oddzielonych spacją. Zatem powyższą instrukcją stworzyliśmy nazwaną krotkę `"tree"` z trzema polami: `species`, `height` oraz `diameter`.
Od tego momentu możemy tworzyć krotki poprzez wywołanie `tree()`, np:

```
t = tree("sosna", 2.45, 0.9)
print(t)
```

Musimy w tym miejscu pamiętać, że tworząc taką krotkę, jesteśmy zobligowani do wypełnienia wszystkich jej pól w takiej kolejności, w jakiej zostały określone.
Po uruchomieniu tego kodu zobaczymy na ekranie tekst w postaci:
tree(species='sosna', height=2.45, diameter=0.9)

Jeżeli chcemy wyświetlić zawartość krotki w bardziej *przyjazny sposób*, możemy wypisać jej elementy samodzielnie:

```
print(t.species, t.height, t.diameter)
```

W efekcie uzyskamy po prostu:
sosna 2.45 0.9

Jak widzisz, do poszczególnych pól drzewa odwołujemy się, podając jego nazwę. Po modyfikacjach kod z zadania 6.2 wygląda zatem następująco:

```
from collections import namedtuple

tree = namedtuple("tree", "species height diameter")
trees = [tree("sosna", 2.01, 35.00), tree("brzoza", 1.98, 29.50),
        tree("klon", 2.50, 31.02), tree("sosna", 2.12, 37.50),
        tree("modrzew", 1.45, 19.56), tree("dąb", 2.35, 50.00),
        tree("świerk", 2.24, 42.01), tree("klon", 1.53, 25.25)]
for t in trees:
    print(t.species, t.height, t.diameter)

highest = trees[0]
for t in trees:
    if t.height>highest.height:
        highest = t
print(highest.species)
```

Rozdział 7

Słowniki

Słownik (ang. *dictionary*) jest strukturą danych podobną do list i krotek, z tą różnicą, że nie pracuje na podstawie indeksów. Słownik to nieuporządkowany zbiór par (*klucz – wartość*). Słowniki nie są sekwencjami, ale odwzorowaniami (). Klucze mogą być obiektem praktycznie dowolnego typu (napisy, liczby, krotki itp.), przy czym muszą być unikalne i tego samego typu dla całego słownika. Wartości w słowniku mogą zawierać dowolny typ danych.

7.1 Zadanie 1 – wyznaczanie częstotliwości liter w tekście

Analiza częstotliwościowa polega na sprawdzaniu, ile razy każda litera pojawia się w zaszyfrowanej wiadomości, i wykorzystaniu tej informacji do złamania kodu. Jeśli litera pojawia się wiele razy w wiadomości, to jest o wiele bardziej prawdopodobne, że jest to "A" niż na przykład "F". Napisz program, który zliczy i wyświetli na ekranie występowanie poszczególnych liter w napisie. Przyjmijmy, że tekst zawiera tylko litery z alfabetu angielskiego.

Zadanie moglibyśmy rozwiązywać na kilka sposobów:

- tworząc *26 zmiennych*, po jednej dla każdej litery alfabetu – każda zmienna byłaby odpowiedzialna za liczność jednej litery,
- tworząc *listę z 26 elementami* – kolejne elementy listy przechowywałyby liczności kolejnych liter alfabetu: pierwszy element – litera "A", drugi element – litera "B" itd.,
- wykorzystując *słownik*, w którym znaki byłyby kluczami, a liczniki odpowiednimi wartościami.

Zaczynamy od wczytania tekstu i stworzenia pustego słownika:

```
text = input("Enter text")
freq = {}
```

Pusty słownik tworzymy, używając nawiasów klamrowych `{}`. Dla przypomnienia, listy tworzyliśmy, korzystając z nawiasów kwadratowych `[]`, a krotki – z nawiasów okrągłych `()`.
Następnie, aby wypełnić słownik, przechodzimy pętlą `for` po napisie:

```
for c in text:
    if c not in freq:
```

```
        freq[c] = 1
    else:
        freq[c] += 1
```

W każdej iteracji sprawdzamy, czy znak zawarty w `c` nie występuje w słowniku: `if c not in freq`. Jeżeli jest to jego pierwsze wystąpienie, to tworzymy nową pozycję z kluczem `c` i wartością początkową 1. Jeśli zaś `c` znajduje się już w słowniku, zwiększamy wartość `freq[c]`.
Jak widzisz, wstawienie pozycji do słownika i odwoływanie się do jego zawartości wygląda tak samo. Słownik, w chwili gdy podany jest nieistniejący klucz, stworzy nowy wpis.
I wyświetlamy słownik na ekranie:

```
print(freq)
```

Pełny kod wygląda następująco:

```
text = input("Enter text")
freq = {}
for c in text:
    if c not in freq:
        freq[c] = 1
    else:
        freq[c] += 1
print(freq)
```

Po uruchomieniu naszego programu możemy zauważyć kilka „problemów”. Po pierwsze, w słowniku znajdą się wszystkie znaki, a nie tylko litery. Po drugie, małe i wielkie litery zajmują osobne pozycje w słowniku.
Pierwszą kwestię możemy rozwiązać, sprawdzając, czy dany znak napisu jest literą, zanim utworzymy pozycję w słowniku:

```
if c.isalpha()==True:
```

drugą – konwertując wszystkie litery na małe (lub wielkie) przy wstawianiu:

```
c.lower()
```

Po modyfikacjach kod będzie wyglądał następująco:

```
text = input("Enter text")
freq = {}
for c in text:
    if c.isalpha()==True:
        if c.lower() not in freq:
            freq[c.lower()] = 1
        else:
```

```
            freq[c.lower()] += 1
print(freq)
```

Naszym celem było obliczenie częstotliwości wystąpień liter, a nie tylko wyznaczenie ich liczności. Musimy zatem zmodyfikować elementy słownika, dzieląc każdą wartość przez liczbę wszystkich liter:

```
count = 0
for key in freq:
    count += freq[key]
```

Sumę wystąpień możemy również policzyć w inny sposób. Wykorzystamy funkcję sumującą elementy listy `sum`. Ale skąd wziąć listę? Słowniki oferują dostęp nie tylko do konkretnych pozycji, lecz także do listy kluczy i listy wartości oddzielnie. Używając metody wbudowanej `.values()`, dla naszego słownika otrzymamy listę samych wartości przechowywanych w słowniku.

```
count = sum(freq.values())
```

W analogiczny sposób moglibyśmy uzyskać listę kluczy – wywołując `freq.keys()`, np.:

```
count = 0
for key in freq.keys():
    count += freq[key]
```

Mając wyznaczoną sumę, możemy wyznaczyć poszczególne częstotliwości:

```
for key in freq:
    freq[key] = freq[key]/count
```

Przy wypisywaniu słownika na pewno zwróciło Twoją uwagę, że klucze są w nim uporządkowane w kolejności ich wstawiania do słownika. A gdybyśmy chcieli mieć dane posortowane alfabetycznie wg kluczy?
Musisz pamiętać, że same słowniki nie podlegają sortowaniu. Ale możemy przekonwertować słownik na listę krotek, posortować ją, a następnie na podstawie posortowanej listy utworzyć słownik.
Korzystając z metody słownika `.items()`, tworzymy listę, w której każdy element zawiera krotkę z kluczem i przypisaną do niego wartością:

```
lfreq = freq.items()
```

tworzymy listę posortowaną:

```
lsfreq = sorted(lfreq)
```

a następnie na jej podstawie tworzymy słownik i wyświetlamy go na ekranie:

```
freq_sorted = dict(lsfreq)
print(freq_sorted)
```

Można to również zapisać w bardziej skondensowanej formie:

```
freq_sorted = dict(sorted(freq.items()))
print(freq_sorted)
```

7.2 Zadanie 2 – wyszukiwanie drzew danego gatunku w lesie

Dana jest lista, w której każdy element to krotka opisująca podstawowe parametry drzewa w lesie: gatunek, wysokość oraz średnicę pnia. Napisz program, który pozwoli na wyświetlenie na ekranie wszystkich drzew gatunku podanego przez użytkownika. Napisz program, który wyświetli nazwę najczęściej pojawiającego się drzewa.

Zaczynamy od stworzenia listy drzew. Wykorzystamy tu poznane w Zadaniu 6.3 krotki nazwane.

```
from collections import namedtuple

tree = namedtuple("tree", "species height diameter")
trees = [tree("sosna", 2.01, 35.00), tree("brzoza", 1.98, 29.50),
        tree("klon", 2.50, 31.02), tree("sosna", 2.12, 37.50),
        tree("modrzew", 1.45, 19.56), tree("dąb", 2.35, 50.00),
        tree("świerk", 2.24, 42.01), tree("klon", 1.53, 25.25)]
```

Następnie tworzymy słownik, w którym będziemy przechowywać pary *gatunek – liczność*. Możemy to zrobić analogicznie do Zadania 7.1:

```
dict_trees = {}
for t in trees:
    if t.species not in dict_trees:
        dict_trees[t.species] = 1
    else:
        dict_trees[t.species] += 1
```

Mając wypełniony słownik, chcemy sprawdzić, który gatunek występuje w nim najczęściej. Wiemy z Zadania 7.1, że słowników się nie sortuje. Aby uzyskać dane w interesującym nas porządku, należy stworzyć na podstawie słownika listę i następnie ją posortować.
Deklarujemy zatem pustą listę:

```
list = []
```

i iterując po słowniku drzew, odczytujemy każdą jego pozycję:

```
for species, count in dict_trees.items():
```

Powyższa pętla ma dwie zmienne iteracyjne, ponieważ `dict_trees.items()` zwraca zbiór par (a dokładniej krotek) w postaci *klucz–wartość*, które przypisujemy w tym przykładzie kolejno do zmiennych `species` oraz `count`.
Dla każdej pozycji słownika tworzymy pozycję na liście:

```
    list.append((count, species))
```

Zwróć uwagę, że krotka umieszczana na liście ma *odwróconą* kolejność elementów – najpierw liczba drzew `count`, a dopiero potem gatunek `species`. Skorzystamy z tego przy sortowaniu listy.

```
    list.sort(reverse=True)
```

W przypadku, gdy lista zawiera krotki, a nie pojedyncze wartości, takie jak liczby, funkcja `.sort()` ustala kolejność elementów na podstawie pierwszego elementu krotki. Skorzystaliśmy z tego w Zadaniu 7.1, gdzie w liście trzymaliśmy krotki w postaci *litera–częstotliwość*. Jednak w tym zadaniu chcemy posortować listę wg liczności występowania drzew. Czyli bardziej istotna jest dla nas *wartość* w słowniku, a nie *klucz*.
Ponieważ liczność jest pierwsza, będzie ona użyta do porównań. Jeśli jest więcej niż jedna krotka o tej samej wartości, to operacja sortowania spojrzy na drugi element (gatunek), więc krotki, w których liczność jest taka sama, będą posortowane w kolejności alfabetycznej po nazwie gatunku.
Wywołanie funkcji `.sort` z parametrem `reversed=True` wskazuje, by sortować w kolejności malejącej.
Pozostało jeszcze wyświetlenie nazw gatunków, które występują najczęściej. Nie wystarczy, że wypiszemy na ekranie pierwszy element listy, ponieważ kilka gatunków może występować w maksymalnej liczności.
Po wyświetleniu pierwszego gatunku na liście musimy zatem przejść listę (zaczynając od jej drugiego elementu, czyli indeksu 1) i sprawdzić, czy kolejne elementy mają liczność równą liczności pierwszego. Jeżeli tak, to wyświetlamy na ekranie jego nazwę. W przeciwnym wypadku przerywamy pętlę, ponieważ lista uporządkowana jest malejąco pod względem liczności:

```
print("The most common tree species are: ")
print(list[0][1])
for i in range(1, len(list)):
    if list[i][0] == list[0][0]:
        print(list[i][1])
    else:
        break
```

Łącząc wszystkie powyższe fragmenty, otrzymujemy następujący kod:

```
from collections import namedtuple
```

```
tree = namedtuple("tree", "species height diameter")
trees = [tree("sosna", 2.01, 35.00), tree("brzoza", 1.98, 29.50),
        tree("klon", 2.50, 31.02), tree("sosna", 2.12, 37.50),
        tree("modrzew", 1.45, 19.56), tree("dąb", 2.35, 50.00),
        tree("świerk", 2.24, 42.01), tree("klon", 1.53, 25.25)]
#print(trees)

dict_trees = {}
for t in trees:
    if t.species not in dict_trees:
        dict_trees[t.species] = 1
    else:
        dict_trees[t.species] += 1
print("Dictionary of trees:")
print(dict_trees)

list = []
for species, count in dict_trees.items():
    list.append((count, species))
list.sort(reverse=True)
print("Sorted list:")
print(list)

print("The most common tree species are: ")
print(list[0][1])
for i in range(1, len(list)):
    if list[i][0] == list[0][0]:
        print(list[i][1])
    else:
        break
```

7.3 Zadanie 3 – wyznaczanie dzielników właściwych liczby

Napisz program, który dla każdej liczby z danego przedziału wyznaczy listę jej dzielników właściwych (mniejszych od tej liczby).

Wynik działania naszego programu umieścimy w słowniku, gdzie kluczem będą kolejne liczby całkowite z przedziału, a wartościami – lista dzielników liczby z klucza.

Zaczynamy od wczytania krańców przedziału i stworzenia pustego słownika:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
div_dict = {}
```

Tworzymy pętlę, która będzie iterowała po kolejnych liczbach z przedziału:

```
for n in range(a, b+1):
```

Dla każdej wartości `n` tworzymy pustą listę dzielników:

```
    divisors = []
```

I sprawdzamy, czy kolejne liczby mniejsze od `n` są jej dzielnikami. Jeśli tak, to uzupełniamy listę dzielników:

```
    for i in range(1, n):
        if n%i == 0:
            divisors.append(i)
```

Tworzymy pozycję w słowniku:

```
    div_dict[n] = divisors
```

Kod programu uzupełniony o wyświetlanie słownika na ekranie prezentuje się następująco:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

div_dict = {}
for n in range(a, b+1):
    divisors = []
    for i in range(1, n):
        if n%i == 0:
            divisors.append(i)
    div_dict[n] = divisors
```

Rozdział 8

Funkcje

Funkcja to wydzielona część programu wykonująca pewne operacje. Nazwa funkcji ma istotne znaczenie, określa ona czynność, którą dana funkcja wykonuje. Poprzez użycie odpowiedniej nazwy można wywołać konkretną funkcję, czyli wykonać kod w niej zapisany.
Tworzenie funkcji w języku *Python* rozpoczynamy od słowa kluczowego `def`. Zaraz po nim następują nazwa funkcji, nawiasy okrągłe, w którym umieszczane są parametry funkcji, oraz dwukropek.
Wartości z funkcji zwracane są za pomocą słowa kluczowego `return`. Wartości zwracane z funkcji nie muszą być tylko liczbami. Funkcje mogą zwracać dowolne obiekty (np. listy, krotki, słowniki).

8.1 Zadanie 1 – obliczanie iloczynu dwóch liczb

Napisz funkcję obliczającą iloczyn dwóch liczb.

Zaczynamy od zdefiniowania nagłówka funkcji. Nazwa funkcji `multiplication` jest poprzedzona słowem kluczowym `def`. Następnie w nawiasach okrągłych określamy parametry funkcji: w tym wypadku dwie wartości `x` i `y`:

```
def multiplication(x, y):
```

Obliczamy iloczyn i zwracamy wynik:

```
    res = x*y
    return res
```

Należy zwrócić uwagę na fakt, że zmienne zadeklarowane wewnątrz funkcji nie są połączone w żaden sposób z innymi zmiennymi o tych samych nazwach, ale użytych w innej części programu. Innymi słowy, te zmienne są *lokalne* dla tej funkcji. Każda zmienna ma swój zakres, czyli blok, w którym została zadeklarowana, zaczynając od miejsca zdefiniowania jej nazwy.
Powyższe dwie instrukcje możemy również połączyć w jedną:

```
    return x*y
```

Kod funkcji wraz z wywołaniem wygląda następująco:

```
def multiplication(x, y):
    return x*y
```

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print(multiplication(a, b))
```

8.2 Zadanie 2 – obliczanie ilorazu dwóch liczb

Napisz funkcję obliczającą iloraz dwóch liczb.

Funkcję obliczającą iloraz dwóch liczb tworzymy analogicznie do wyznaczającej iloczyn:

```
def division(x, y):
    return x/y
```

A jej wywołanie:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print(division(a, b))
```

Zastanówmy się jednak, co w przypadku, gdy wartość drugiego parametru będzie równa 0? Wiadomo, że wynik takiego dzielenia nie istnieje. Co w takim wypadku zwróci funkcja?
Po uruchomieniu okaże się, że program zgłosił błąd w postaci: *ZeroDivisionError: division by zero*. Musimy zatem wykluczyć w naszej funkcji wartość 0 w mianowniku:

```
def division(x, y):
    if y!=0:
        return x/y
```

W obecnej postaci zwracamy wynik tylko wtedy, gdy wykonanie dzielenia jest możliwe. Ale co się dzieje, gdy funkcja dostanie 0 w drugim parametrze? Funkcja `print()` wyświetli na ekranie `None`. `None` to specjalny typ, który reprezentuje po prostu *nic* (wartość `None` należy do typu danych o nazwie `NoneType`). Każda funkcja, w której nie skorzystamy z `return`, domyślnie zwraca właśnie `None`. Również w wypadku, gdy nie użyliśmy `return` we wszystkich możliwych *ścieżkach zakończenia* funkcji (a tak jest w naszym przypadku), zwracana jest wartość `None`.
Moglibyśmy zostawić funkcję w tej postaci, gdyż uporaliśmy się już z problemem dzielenia przez 0 – nasza funkcja nie dopuści do takiego działania. Wyobraźmy sobie jednak, że wynik naszej funkcji jest częścią dalszych obliczeń, dla uproszczenia wykonajmy po prostu:

```
quotient = division(a, b)
print(quotient+2)
```

Po przekazaniu `b==0` okaże się, że program zakończył się kolejnym błędem: *unsupported operand type(s) for +: 'NoneType' and 'int'*.
Do przedstawiania brakujących danych liczbowych, a tak będzie w naszym przypadku, lepiej od `None` nadaje się wartość `nan`. W przeciwieństwie do `None` `nan` jest wartością typu liczbowego (a dokładniej mówiąc, zmiennopozycyjnego), więc jej użycie w rachunkach nie powoduje błędu. Jakiekolwiek działanie, w którym występuje `nan`, zawsze zwraca wynik `nan`.
Pozostaje nam zatem uzupełnienie funkcji o kolejną instrukcję:

```
import math

def division (x, y):
    if y!=0:
        return x/y
    return math.nan
```

Musimy również zaimportować bibliotekę `math`, w której `nan` jest zdefiniowane.
Obecnie wywołanie:

```
print(quotient+2)
```

pokaże na ekranie wartość `nan`.

Zmodyfikujmy również wywołanie naszej funkcji:

```
quotient = division(a, b)
if (not math.isnan(quotient)):
    print(quotient)
else:
    print('Division by 0!')
```

Funkcja `math.isnan()` służy do sprawdzania, czy dana zmienna ma wartość `nan` czy nie. Do tego sprawdzenia nie możemy użyć po prostu operatora porównania `==`. `nan` nie jest równe niczemu (nawet sobie!).
W tym miejscu należy doprecyzować, że `nan` nie jest tak naprawdę liczbą. Jest numerycznym typem danych używanym do reprezentowania dowolnej wartości, która jest niezdefiniowana lub nieprzedstawialna.
Na przykład $0/0$ jest niezdefiniowane jako liczba rzeczywista i dlatego jest przedstawiane przez `nan`. Pierwiastek kwadratowy z liczby ujemnej jest liczbą urojoną, która nie może być określona jako liczba rzeczywista, jest zatem również prezentowany przez `nan`.
Zapamiętaj, że `nan` jest także przypisywane do zmiennych w obliczeniach, które nie mają wartości i nie zostały jeszcze obliczone.

8.3 Zadanie 3 – obliczanie sumy elementów listy

Napisz funkcję zwracającą sumę elementów na liście.

Nagłówek naszej funkcji wygląda następująco:

```
def sum_of_elements(numbers):
```

Następnie deklarujemy zmienną, która będzie wynikiem sumowania:

```
    sum = 0
```

W pętli dodajemy kolejne elementy listy:

```
    for x in numbers:
        sum += x
```

I zwracamy obliczony wynik:

```
    return sum
```

Oczywiście ta funkcja będzie działała poprawnie tylko wtedy, gdy elementami listy będą wartości liczbowe.
Kod funkcji wraz z wywołaniem będzie wyglądał następująco:

```
def sum_of_elements(numbers):
    sum = 0
    for x in numbers:
        sum += x
    return sum

list = [1, 2, 3, 4]
print(sum_of_elements(list))
```

8.4 Zadanie 4 – sprawdzanie, czy liczba jest liczbą pierwszą

Napisz funkcję sprawdzającą, czy dana liczba jest liczbą pierwszą.

Liczbą pierwszą jest liczba naturalna większa od 1, która ma dokładnie dwa dzielniki naturalne: jedynkę i siebie samą.
Parametrem naszej funkcji będzie liczba `n`:

```
def is_prime(n):
```

Zanim zaczniemy szukać dzielników, możemy wykluczyć pewne przypadki:

```
    if n == 2:
        return True
```

Jeżeli badaną liczbą jest 2, możemy od razu wyjść z funkcji z wartością `True`. W kolejnej instrukcji warunkowej wykluczymy liczby, które nie są naturalne (czyli mniejsze równe 1) oraz wartości parzyste:

```
    if n <= 1 or n % 2 == 0:
        return False
```

Kolejność powyższych instrukcji warunkowych ma znaczenie. Ponieważ liczba 2 jest jedyną parzystą liczbą pierwszą, więc warunek `n == 2` musi być sprawdzony przed badaniem parzystości parametru. W przeciwnym wypadku oznaczylibyśmy 2 jako liczbę, która nie jest pierwsza.
Następnie w pętli szukamy dzielników. Sprawdzamy wszystkie wartości całkowite od 3 (ponieważ 2 już wykluczyliśmy). Zakres poszukiwania dzielników liczby można ograniczyć z góry przez $\sqrt{n}$, ponieważ jest to ostatnia liczba, która może być dzielnikiem liczby `n`. Warto jednak zwrócić uwagę na to, że pierwiastek liczby `n` też może być dzielnikiem liczby `n`, dlatego tym razem przedział z prawej strony musi być domknięty – stąd `+1` przy wyznaczaniu wartości zmiennej `sqrt()`.

```
    sqrt = int(n**0.5) + 1
    for divisor in range(3, sqrt, 2):
```

Krok pętli ustawiamy na 2, aby pominąć parzyste dzielniki.
Jeśli reszta z dzielenia (modulo) jest równa 0, to znaczy, że liczba jest złożona. Przerywamy funkcję, zwracając `False`, bo nie ma potrzeby szukać następnych dzielników.

```
        if n % divisor == 0:
            return False
```

Po zakończeniu pętli wiemy, że badana liczba jest liczbą pierwszą, i w takim wypadku kończymy funkcję z wynikiem `True`:

```
    return True
```

Kod całej funkcji prezentuje się zatem następująco:

```
def is_prime(n):
    if n == 2:
        return True
    if n % 2 == 0 or n <= 1:
        return False

    sqr = int(n**0.5) + 1
    for divisor in range(3, sqr, 2):
```

```
        if n % divisor == 0:
            return False
    return True
```

8.5 Zadanie 5 – wyznaczanie liczb pierwszych z przedziału

Napisz funkcję zwracającą listę zawierającą liczby pierwsze z danego przedziału.

W tym zadaniu wykorzystamy funkcję `is_prime` napisaną w Przykładzie 8.4. Zaczynamy od zdefiniowania nagłówka funkcji. Parametry `a` i `b` oznaczają odpowiednio początek i koniec przedziału:

```
def prime_numbers(a, b):
```

Na wszelki wypadek sprawdzamy, czy kolejność krańców przedziału jest prawidłowa. Jeśli nie, to zamieniamy wartości:

```
    if a>b:
        a, b = b, a
```

Tworzymy pustą listę, w której będziemy umieszczać liczby pierwsze:

```
     numbers = []
```

W pętli iterujemy po liczbach w podanym zakresie:

```
    for n in range(a, b+1):
```

Drugi parametr funkcji `range` jest równy `b+1`, ponieważ `range` tworzy przedział prawostronnie otwarty, a my nie chcemy pominąć wartości `b` w poszukiwaniach. Korzystając z napisanej wcześniej funkcji, sprawdzamy, czy bieżąca wartość `n` jest liczbą pierwszą. Jeżeli tak, to dopisujemy ją do listy:

```
        if is_prime(n):
            numbers.append(n)
```

Na koniec zwracamy wypełnioną listę:

```
    return numbers
```

Kod funkcji wraz z przykładowym wywołaniem będzie wyglądał następująco:

```
def prime_numbers(a, b):
    if a>b:
        a, b = b, a
    numbers = []
    for n in range(a, b+1):
```

```
        if is_prime(n):
            numbers.append(n)
    return numbers

l = prime_numbers(1, 1)
if len(l) == 0:
    print("There is no prime number in given interval.")
else:
    print(f"Prime number in interval: {l}")
```

8.6 Zadanie 6 – wyznaczanie liczb bliźniaczych z przedziału

Napisz funkcję wyznaczającą pary liczb bliźniaczych z danego przedziału. Liczby bliźniacze to para liczb pierwszych, których różnica wynosi 2, np.: 3 i 5, 5 i 7, 11 i 13.

Pary liczb bliźniaczych zwrócimy z napisanej funkcji w postaci listy krotek.

Zaczynamy od zdefiniowania nagłówka funkcji. Parametry `a` i `b` oznaczają odpowiednio początek i koniec przedziału:

```
def prime_numbers(a, b):
```

Korzystając z napisanej w Przykładzie 8.5 funkcji, tworzymy listę liczb pierwszych z analizowanego przedziału:

```
    primes = prime_numbers(a, b)
```

Tworzymy również pustą listę, w której umieścimy znalezione liczby bliźniacze:

```
    twins = []
```

Dla każdej wyznaczonej liczby pierwszej:

```
    for i in range(1, len(primes)):
```

sprawdzamy, czy różnica jej samej i jej poprzedniczki na liście jest równa 2:

```
        if primes[i] - primes[i-1] == 2:
```

jeżeli tak, to tworzymy krotkę o nazwie `pair` z tymi liczbami i wstawiamy ją na listę:

```
            pair = (primes[i-1], primes[i])
            twins.append(pair)
```

Zwracamy wypełnioną listę:

```
tp = twin_primes(a, b)
```

Pełny kod funkcji wraz z przykładowym wywołaniem prezentuje się następująco:

```
def twin_primes(a, b):
    primes = prime_numbers(a, b)
    twins = []
    for i in range(1, len(primes)):
        if primes[i] - primes[i-1] == 2:
            pair = (primes[i-1], primes[i])
            twins.append(pair)
            print(f"{primes[i-1]} -- {primes[i]}")
    return twins

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
tp = twin_primes(a, b)
if len(tp) == 0:
    print("There are no twin primes in given interval. ")
else:
    print(f"Twin primes in <{a}, {b}> are: {tp}")
```

8.7 Zadanie 7 – wyznaczanie liczb zaprzyjaźnionych z przedziału

Napisz funkcję zwracającą listę liczb zaprzyjaźnionych z danego przedziału.

Liczby zaprzyjaźnione to para różnych liczb całkowitych dodatnich, w których suma dzielników właściwych każdej z tych liczb równa się drugiej liczbie. Pierwszą parę takich liczb podał Pitagoras, są to 220 i 284.

Nasz program podzielimy na kilka etapów. W pierwszej kolejności stworzymy funkcję znajdującą wszystkie dzielniki danej liczby. Następnie dla wszystkich liczb z podanego przez użytkownika przedziału zdefiniujemy słownik dzielników – podobnie jak w Przykładzie 7.3. Ostatnim krokiem będzie przeszukanie słownika.

Funkcja zwracająca listę dzielników danej liczby:

```
def find_divisors(n):
    divisors = []
    for i in range(1, n):
```

```python
        if n%i == 0:
            divisors.append(i)
    return divisors
```

Funkcja tworząca słownik postaci *liczba – lista dzielników*:

```python
def create_dict(a, b):
    div_dict = {}
    for n in range(a, b+1):
        divisors = find_divisors(n)
        div_dict[n] = divisors
    return div_dict
```

Schemat tworzenia słownika dzielników opisany został w Przykładzie 7.3.

Przystępujemy teraz do tworzenia głównej funkcji naszego zadania.
Definiujemy nagłówek funkcji oraz niezbędne zmienne:

```python
def find_amicable_numbers(start, stop):
    amicables = []
    div_dict = create_dict(start, stop)
    numbers = range(start, stop+1)
```

Parametry funkcji `start` oraz `stop` określają krańce przeszukiwanego przedziału. Pusta obecnie lista `amicables` będzie zawierała pary liczb zaprzyjaźnionych. `div_dict` jest słownikiem dzielników, a lista `numbers` przechowuje kolejne wartości z przedziału. Moglibyśmy również pobrać listę kluczy z naszego słownika:

```python
    numbers = list(div_dict.keys())
```

Korzystając z pętli `for`, przechodzimy kolejno po wartościach z przedziału. Dodatkowo korzystamy tutaj z funkcji `enumerate`, ponieważ będą nam potrzebne nie tylko kolejne wartości `n`, lecz także ich położenie na liście `i`.

```python
    for i, n in enumerate(numbers):
```

Dla każdej wartości `n` wyznaczamy sumę jej dzielników:

```python
        n_div = sum(div_dict[n])
```

Użycie `div_dict[n]` zwraca nam wartość ze słownika przypisaną kluczowi `n`.
Funkcja `sum` wyznacza sumę elementów na liście.
Następnie sprawdzamy każdą liczbę z przedziału większą od aktualnej wartości zmiennej `n`:

```python
        for j in range(i+1, len(numbers)):
```

Pobieramy jej wartość (`j` jest pozycją, więc `numbers[j]` daje nam kolejną liczbę z przedziału) oraz wyznaczamy sumę dzielników, analogicznie jak w przypadku zmiennej `n`:

```
            b = numbers[j]
            b_div = sum(div_dict[b])
```

Sprawdzając warunek o równości zmiennych i sum dzielników, decydujemy, czy parę liczb n, b dopisać do listy liczb zaprzyjaźnionych:

```
            if n == b_div and b == n_div:
                amicables.append((n, b))
```

Należy pamiętać o ujęciu n, b w dodatkowe nawiasy okrągłe – zależy nam na stworzeniu krotki.
Na koniec zwracamy jeszcze stworzoną listę:

```
    return amicables
```

Łącząc powyższe funkcje, otrzymujemy następujący kod:

```
def find_amicable_numbers(start, stop):
    div_dict = create_dict(start, stop)
    amicables = []
    numbers = list(div_dict.keys())
    for i, n in enumerate(numbers):
        n_div = sum(div_dict[n])
        for j in range(i+1, len(numbers)):
            b = numbers[j]
            b_div = sum(div_dict[b])
            if n == b_div and b == n_div:
                amicables.append((n, b))
    return amicables

def find_divisors(n):
    divisors = []
    for i in range(1, n):
        if n%i == 0:
            divisors.append(i)
    return divisors

def create_dict(a, b):
    div_dict = {}
    for n in range(a, b+1):
        divisors = find_divisors(n)
        div_dict[n] = divisors
```

```
    return div_dict

def find_amicable_numbers(start, stop):
    div_dict = create_dict(start, stop)
    amicables = []
    numbers = list(div_dict.keys())
    for i, n in enumerate(numbers):
        n_div = sum(div_dict[n])
        for j in range(i+1, len(numbers)):
            b = numbers[j]
            b_div = sum(div_dict[b])
            if n == b_div and b == n_div:
                amicables.append((n, b))
    return amicables

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print(find_amicable_numbers(a, b))
```

Rozdział 9

Pliki

Do obsługi plików potrzebujemy specjalnej zmiennej, która będzie pełniła funkcję uchwytu do pliku. Wszelkie operacje związane z plikiem wykonuje się właśnie poprzez metody oferowane przez zmienną plikową.

Otwarcie pliku

Do otwierania pliku służy metoda `.open()`. Zwraca uchwyt do pliku, a w parametrze przyjmuje ścieżkę do pliku i tryb, w którym chcemy pracować z plikiem:

```
f = open("directory/file_path.txt", "mode")
```

Ścieżka może być bezwzględna (cała ścieżka do pliku) lub może to być ścieżka z miejsca, w którym uruchomiony jest program. Jeśli uruchamiasz program w folderze, w którym znajduje się plik, wystarczy po prostu wpisać jego nazwę wraz z rozszerzeniem.
Drugim parametrem metody jest tryb otwarcia pliku. Najczęściej stosowane tryby to:

- `"r"` (*read*) – plik otwarty do odczytu (próba otwarcia pliku, którego nie ma w podanej lokalizacji, kończy się błędem),
- `"w"` (*write*) – plik otwarty do zapisu (jeżeli pliku nie ma w podanej lokalizacji, zostanie on utworzony, przed zapisem zawartość pliku jest usuwana),
- `"a"` (*append*) – plik otwarty do zapisu (jeżeli pliku nie ma w podanej lokalizacji, zostanie on utworzony, nowa treść jest dodawana na końcu pliku, bez usuwania starej).

Drugi argument jest opcjonalny. Jeśli go nie podamy, plik otwierany jest w trybie do odczytu `"r"`. Domyślnie wszystkie pliki otwierane są w trybie tekstowym.

Zamknięcie pliku

Otwarty plik należy zamknąć. Służy do tego metoda `.close()`:

```
f.close()
```

Należy o tym pamiętać, ponieważ czasem może dojść do sytuacji, w której niezamknięty plik ulegnie uszkodzeniu lub będzie blokowany, podczas gdy inny program będzie chciał uzyskać do niego dostęp.

Czytanie z pliku

Tak otwarty plik można przeczytać, używając metody `.read()` wywoływanej z utworzonej zmiennej plikowej:

```
f = open("numbers.txt", "r")
content = f.read()
```

Po tej operacji zawartość pliku zapisaną w zmiennej `content` traktujemy jako jeden wieloliniowy napis.
Możliwe jest odczytanie pojedynczej linii pliku i umieszczenie jej w napisie `line`:

```
f = open("numbers.txt", "r")
line = f.readline()
```

Warto pamiętać, że metoda `.readline()` wczytuje też *Enter* kończący linię.
Jak również wczytanie całej zawartości pliku w liście napisów `lines`:

```
f = open("numbers.txt", "r")
lines = f.readlines()
```

Pojedyncze linie zawierają kończący je w pliku *Enter*.

Zapis do pliku

Do zapisu do pliku służy metoda `.write()`.

```
f = open("out_numbers.txt", "w")
f.write("First line.\n")
```

Wartość podana do zapisu musi być napisem. Jeśli chcemy, aby w pliku znalazł się znak nowej linii, musimy wstawić go samodzielnie, dodając, jak w przykładzie, znak `""` do wpisywanej linii.
Do pliku możemy zapisać jednocześnie wiele linii. W tym celu wykorzystamy listę napisów:

```
f = open("out_numbers.txt", "w")
lines = ["First line.\n", "Second line.\n", "Third line.\n"]
f.writelines(lines)
```

Ponownie każdy napis uzupełniamy znakiem *Enter*.
Plik otworzony w trybie `"w"` przy otwarciu zostaje wyczyszczony. Jeżeli chcemy dopisać nowe dane do starych, używamy trybu `"a"`.

```
f = open("out_numbers.txt", "w")
lines = ["Fourth line.\n", "Fifth line.\n"]
f.writelines(lines)
```

Struktura `with`

Do otwarcia pliku możemy również wykorzystać alternatywną metodę, która nie wymaga dodatkowego zamykania pliku. Jest to możliwe dzięki wykorzystaniu struktury `with`:

```
lines = []
with open("data.txt", "r") as f:
    lines = f.readlines()
lines.sort()
with open("out_data.txt", "w") as f:
    f.writelines(lines)
```

W takim przypadku plik zostanie automatycznie zamknięty, tak jakbyśmy wywołali metodę `.close()`.
Python pozwala również w ten sposób otworzyć plik zarówno do zapisu, jak i odczytu w jednej strukturze:

```
with open("numbers.txt", "r") as rfile,
        open("out_numbers.txt", "w") as wfile:
    numbers = rfile.readlines()
    for n in numbers:
        if int(n)%2 == 0:
            wfile.write(n)
```

9.1 Zadanie 1 – rozdzielenie wartości

Dany jest plik zawierający liczby całkowite. Napisz funkcję, która rozdzieli zawartość tego pliku do dwóch kolejnych: w pierwszym znajdą się wszystkie wartości mniejsze od średniej z pierwszego pliku, w drugim – większe bądź równe.

Dane umieściliśmy w pliku *"numbers.txt"* znajdującym się w tym samym katalogu co kod naszego programu. Po nawiązaniu połączenia z plikiem:

```
with open("numbers.txt", "r") as rfile:
```

wczytujemy jego zawartość do listy `numbers`:

```
    numbers = rfile.readlines()
```

Obliczamy średnią arytmetyczną wartości na liście:

```
    avg = 0
    for n in numbers:
        avg += int(n)
    avg /= len(numbers)
```

Należy pamiętać, że elementami listy `numbers` są napisy, stąd konieczne jest użycie funkcji `int()`.

W kolejnym kroku otwieramy dwa pliki do zapisu. Jeżeli w katalogu bieżącym są już pliki o tej nazwie, to ich zawartość zostanie nadpisana. W przeciwnym wypadku zostaną utworzone. W pliku o nazwie *"lower_numbers.txt"* znajdą się wartości mniejsze od średniej arytmetycznej, a w *"greater_numbers.txt"* pozostałe:

```
    with open("lower_numbers.txt", "w") as wfile_l,
            open("greater_numbers.txt", "w") as wfile_g:
```

Przechodzimy listę z wartościami i sprawdzamy, do którego pliku ją wpisać:

```
        for n in numbers:
            if int(n)<avg:
                wfile_l.write(n)
            else:
                wfile_g.write(n)
```

Kod całego programu wygląda następująco:

```
with open("numbers.txt", "r") as rfile:
    numbers = rfile.readlines()
    avg = 0
    for n in numbers:
        avg += int(n)
    avg /= len(numbers)
    print(avg)
    with open("lower_numbers.txt", "w") as wfile_l,
            open("greater_numbers.txt", "w") as wfile_g:
        for n in numbers:
```

```
            if int(n)<avg:
                wfile_l.write(n)
            else:
                wfile_g.write(n)
```

9.2 Zadanie 2 – wyszukiwanie towarów o największej podwyżce cen

Dany jest plik tekstowy, w którym każda linia zawiera nazwę towaru oraz jego cenę w styczniu i czerwcu. Poszczególne wartości oddzielone są spacją. Napisz funkcję, która znajdzie towary, które podrożały najwięcej, i ich nazwy zapisze do pliku wynikowego.

Przykład:

chleb	3.02	4.15
mleko	2.89	3.99
ser	6.99	8.99
pomidor	4.99	6.99
papryka	9.98	11.50
awokado	6.25	7.99
gruszka	3.50	5.20
kiwi	8.25	9.55

Zanim przeczytamy zawartość pliku, zastanówmy się nad sposobem przechowywania danych. Moglibyśmy, wzorem poprzedniego zadania, wstawić je na listę w takiej formie, w jakiej występują w pliku. W dalszych krokach szukalibyśmy największej różnicy.

Możemy również w czasie wczytywania przygotować dane w taki sposób, by uprościć szukanie maksimum. Wystarczy, że będziemy przechowywać nazwę towaru oraz różnicę w cenie między czerwcem a styczniem – stworzymy w tym celu listę krotek. Aby jeszcze bardziej ułatwić dalsze kroki, elementy w krotkach będziemy przechowywać w kolejności *różnica cen – nazwa*. Umożliwi nam to późniejsze posortowanie listy malejąco po różnicy w cenach, tak jak w Zadaniu 7.2.

Po otwarciu pliku z danymi:

```
with open("prices.txt") as rfile:
```

tworzymy pustą listę.

```
    prices = []
```

Następnie tworzymy pętlę odczytującą dane z pliku:

```
for p in rfile:
```

Zwróć uwagę na zapis `for p in rfile`. Przypomina pętle wykorzystane przy napisach czy też listach. W *Pythonie* po pliku możemy iterować tak samo jak po innych typach sekwencyjnych. Kolejnymi wartościami zmiennej `p` będą następne wiersze w pliku wraz z kończącym je *Enterem*.
Z danej linii wydobywamy poszczególne elementy:

```
p2 = p.strip()
product = p2.split(" ")
```

Metoda `.strip()` usuwa białe znaki z końca linii, czyli znaki spacji i znaki końca linii (*Enter*). Tak zmodyfikowaną linię przypisujemy do zmiennej `p2`.
Następnie za pomocą metody `.split(" ")` z jednego długiego napisu robimy listę wyrazów. W naszym wypadku będzie to lista trzyelementowa zawierająca kolejno nazwę towaru, cenę w styczniu i cenę w czerwcu. Parametrem metody `.split()` jest napis z separatorami, którymi oddzielone są kolejne pozycje. Domyślnym separatorem jest spacja, więc w naszym przypadku możemy go pominąć przy wywołaniu. Te dwie instrukcje można połączyć, zapisując jako:

```
product = p.strip().split()
```

Mając dostęp do poszczególnych elementów w linii, tworzymy krotkę i umieszczamy ją na liście:

```
prices.append( (float(product[2]) - float(product[1]),
               product[0]) )
```

Zauważ, że konieczne jest użycie funkcji `float()`. Przed obliczeniem różnicy musimy zamienić ceny na wersję liczbową.
Po zakończeniu pętli sortujemy listę, podając parametr `reverse=True`, co da nam porządek malejący:

```
prices.sort(reverse=True)
```

Otwieramy plik do zapisu:

```
with open("out_prices.txt", "w") as wfile:
```

i wpisujemy do niego pierwszą pozycję z listy, pamiętając o dopisaniu znaku końca linii:

```
wfile.write(prices[0][1]+"\n")
```

Po posortowaniu w kolejności malejącej na pierwszej pozycji znajduje się towar o największej różnicy. Nie możemy jednak na tym poprzestać. Podobnie jak w Zadaniu 7.2, musimy sprawdzić, czy kolejne towary nie mają takiej samej różnicy w cenach. W tym celu skonstruujemy pętlę:

```
        i = 1
        while(prices[i][0]==prices[0][0]):
            wfile.write(prices[i][1]+"\n")
            i+=1
```

która będzie się wykonywać tak długo, jak długo różnica cen kolejnych pozycji na liście będzie równa pierwszej różnicy, czyli maksymalnej różnicy.

Pełny kod ma zatem następującą postać:

```
with open("prices.txt") as rfile:
    prices = []
    for p in rfile:
        product = p.strip().split(' ')
        prices.append( (float(product[2]) - float(product[1]),
                        product[0]) )
    prices.sort(reverse=True)

    with open("out_prices.txt", "w") as wfile:
        wfile.write(prices[0][1]+"\n")
        i = 1
        while(prices[i][0]==prices[0][0]):
            wfile.write(prices[i][1]+"\n")
            i+=1
```

9.3 Zadanie 3 – podsumowanie ocen

W pliku tekstowym zapisane są dane o osobach i ich ocenach. Elementy w linii oddzielone są od siebie dwoma znakami: przecinkiem i spacją. Do pliku wynikowego należy wpisać, ile jakich ocen wystawiono.

Przykład:

Plik wejściowy:			Plik wyjściowy:
Kowalski,	Jan,	3	5.0 – 2
Nowak,	Bartosz,	4	4.5 – 0
Kokoszko,	Julia,	3.5	4.0 – 1
Wnorowski,	Krzysztof,	5	3.5 – 2
Jakubowska,	Aleksandra,	5	3.0 – 2
Nowicki,	Tomasz,	2	2.0 – 1
Akacki,	Jan,	3	
Babacki,	Karol,	3.5	

Zadanie to rozwiążemy przy pomocy słownika, w którym kluczem będą oceny, a wartościami liczba wystąpień każdej oceny.

Standardowo zaczynamy od otworzenia pliku z danymi:

```
with open("notes.txt") as rfile:
```

Następnie tworzymy słownik z zerowymi wartościami. Dlaczego robimy to wcześniej, a nie w trakcie wczytywania danych? Po pierwsze, w tym wypadku wiemy, jakie klucze wystąpią w słowniku. Po drugie, co ważniejsze, nie każda ocena musi wystąpić w danych (jak w przykładowych danych w treści zadania). Gdybyśmy pozycje słownika tworzyli tylko na podstawie danych z pliku, to niektóre oceny mogłyby zostać pominięte.

```
    data = {5.0: 0, 4.5: 0, 4.0: 0, 3.5: 0, 3.0: 0, 2.0: 0}
```

Czytamy kolejne linie z pliku, rozdzielamy je na listę elementów i modyfikujemy pozycję słownika. Klucze w słowniku są liczbami rzeczywistymi, zatem ponownie używamy `float()`, by zamienić napis na wartość liczbową:

```
    for line in rfile:
        person = line.strip().split(", ")
        data[float(person[2])] += 1
```

Tym razem nie pomijamy parametru w metodzie `.split()`. Gdybyśmy to zrobili, to zarówno w nazwisku, jak i imieniu studenta znalazłyby się znaki przecinka. Ponieważ dane wartości w linii oddzielone są ", ", taką też wartość parametru wysyłamy do `.split()`.
Mając wypełniony słownik, tworzymy na jego podstawie listę i sortujemy ją malejąco po kluczach:

```
    notes = list(data.items())
    notes.sort(reverse=True)
```

Dla przypomnienia, `items()` zwraca listę krotek w postaci (*klucz – wartość*).
Po otwarciu pliku do zapisu:

```
    with open("out_notes.txt", "w") as wfile:
```

wstawiamy do niego kolejne pozycje z listy:

```
        for n in notes:
            wfile.write(f"{n[0]:1}    {n[1]}\n")
```

Zbierając wszystkie informacje w całość, otrzymamy poniższy kod:

```
with open("notes.txt") as rfile:
    data = {5.0: 0, 4.5: 0, 4.0: 0, 3.5: 0, 3.0: 0, 2.0: 0}
```

```
for line in rfile:
    person = line.strip().split(" ")
    data[float(person[2])] += 1
print(data)

notes = list(data.items())
notes.sort(reverse=True)
print(notes)

with open("out_notes.txt", "w") as wfile:
    for n in notes:
        wfile.write(f"{n[0]:1}    {n[1]}\n")
```

Podsumowanie

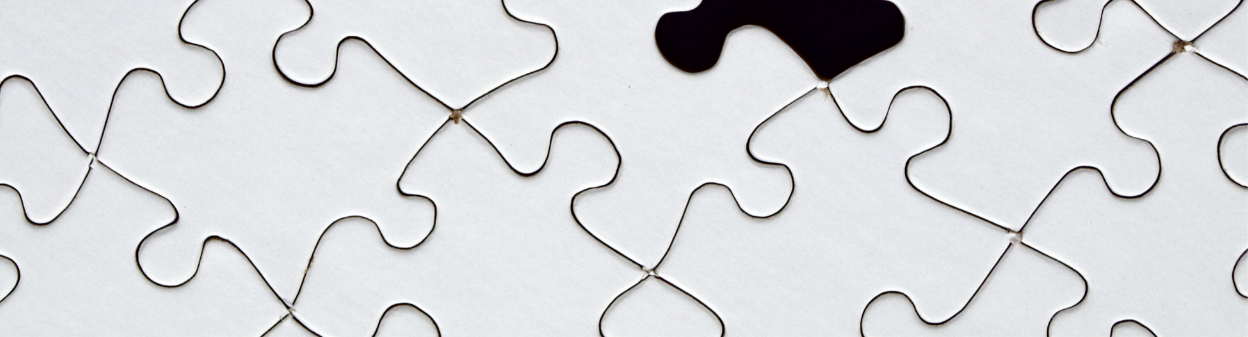
Niniejszy skrypt zbliża się do końca. Stanowi on zaledwie wstęp do programowania w języku *Python*. Omawia podstawowe zagadnienia, takie jak instrukcje wejścia–wyjścia, instrukcje warunkowe, instrukcje iteracyjne, typ napisowy, listy, krotki, słowniki, funkcje oraz obsługę plików. Każdy rozdział, poza krótkim wstępem teoretycznym, zawiera liczne zadania ze szczegółowo omówionymi rozwiązaniami.

Jednakże poznawanie *Pythona* warto kontynuować. *Python* jest językiem wysokiego poziomu, co oznacza, że programowanie w nim jest prostsze i bardziej intuicyjne niż w innych językach programowania. Według raportu *Stack Overflow* z maja 2021 roku *Python* zajął trzecią pozycję wśród najpopularniejszych technologii na świecie. Jest szeroko stosowany w tworzeniu aplikacji internetowych – frameworki, takie jak *Django* i *Flask*, oferują łatwe w użyciu narzędzia do tworzenia webowych aplikacji. *Python* może być również używany do tworzenia aplikacji desktopowych z wykorzystaniem frameworków takich jak *PyQt* i *Tkinter*. *Python* jest używany w tworzeniu gier, szczególnie ich prototypów, a także, dzięki łatwości użycia i czytelności składni, w automatyzacji zadań, takich jak zbieranie i przetwarzanie danych z różnych źródeł, przetwarzanie plików itp. Jego popularność będzie nadal rosła z uwagi na wykorzystanie go w dynamicznie rozwijających się branżach nauki o danych, uczenia maszynowego i sztucznej inteligencji.

Teraz czas na samodzielną pracę – szczególnie nad projektami opartymi na własnych pomysłach.

Powodzenia!

Anna Łupińska-Dubicka
Andrzej Chmielewski



Politechnika
Białostocka