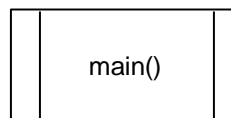


Tworzenie własnych funkcji w Pythonie

Funkcja (w niektórych językach programowania zwana też procedurą) jest fragmentem kodu, który wykonuje ściśle określone działania (najczęściej obliczeniowe, ale nie tylko), przy czym z wnętrza jednej funkcji można wywoływać inną i tak dalej. Podzielenie kodu na funkcje ma wiele zalet. Dzięki nim otrzymujemy m.in. prostszy kod do wielokrotnego wykorzystania w tym samym programie, umożliwiając skuteczniejsze testowanie i debugowanie (każdą funkcję można testować osobno), szybsze tworzenie oprogramowania oraz łatwiejszą pracę w zespołach programistycznych.

Funkcje w programie spełniają rolę podobną do funkcji matematycznych. Dla przykładu, zadaniem funkcji $f(x) = x^2$ jest obliczenie konkretnej wartości wyrażenia po drugiej stronie znaku równości, pod warunkiem, że dostarczymy funkcji odpowiedni argument. Wówczas np. dla argumentu 5 funkcja obliczy wartość $f(5) = 25$.

Na schematach blokowych funkcję lub procedurę (np. o nazwie `main`) przedstawia się za pomocą następującego bloku:



Funkcja wyposażona jest najczęściej w parametry, które oczekują na argumenty przekazywane z innych miejsc programu. Parametry przypominają tutaj stałe (i numerowane) miejsca parkingowe, na których w różnych porach mogą parkować różne samochody, co odpowiada argumentom. Jeśli funkcja ma za zadanie zwrócić jakąś wartość, musi otrzymać jakieś dane (argumenty) oraz musi być wyposażona w instrukcję zwrotną **return**, która kończy jej działanie. Funkcja może być wyposażona w bardzo wiele parametrów formalnych dowolnego typu, do których zostaną przekazane argumenty.

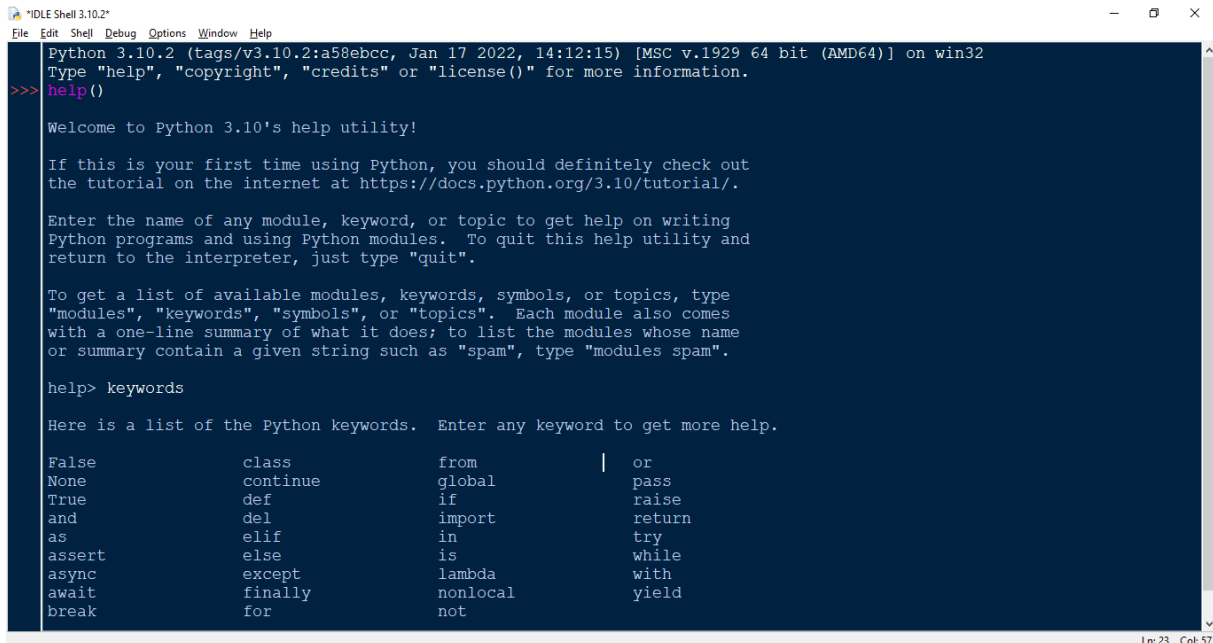
Opisany rodzaj programowania nazywany jest często **programowaniem proceduralnym**, które polega na tworzeniu programu składającego się z wielu funkcji i procedur. W ten sposób cały kod programu składa się najczęściej z wielu funkcji, które ściśle ze sobą współpracują. Jest to krok w stronę **programowania zorientowanego obiektowo** (ang. *Object Oriented Programming* – OOP) lub w skrócie **programowania obiektowego**, czyli najczęściej stosowanego w praktyce paradygmatu (wzorca) programowania. Ostatnio często stosuje się też **programowanie czysto funkcyjne**, które polega na tworzeniu małych funkcji i składaniu ich w programie w coraz bardziej skomplikowane wyrażenia funkcyjne.

Użycie (wywołanie) własnej funkcji w języku Python (w innych językach również) wymaga jej uprzedniego zdefiniowania. Polega ono na użyciu słowa kluczowego `def`, po którym następuje nazwa funkcji z parą okrągłych nawiasów (w których mogą znajdować się parametry) i dwukropkiem. W ciele funkcji (zwanej blokiem) znajdują się instrukcje do wykonania.

```
def nazwa_funkcji(): #nagłówek funkcji (bez parametrów)
    #początek bloku instrukcji
    instrukcja
    instrukcja
    .....
    #koniec bloku instrukcji
    return #jeśli funkcja ma zwracać jakąś wartość
```

W nazewnictwie funkcji obowiązują te same zasady, co w nazewnictwie zmiennych. Najlepiej, gdy nazwa funkcji jest dobrze skojarzona z tym, co funkcja faktycznie wykonuje, albowiem w wielu sytuacjach

znakomicie ułatwia to analizę kodu. Należy przy tym pamiętać, że jako nazwa nie może być wykorzystywane słowo kluczowe języka Python. Nowe wersje Pythona dokładają czasami kolejne słowa kluczowe (aktualnie jest ich 35). Łatwo zapoznać się z ich wykazem, wywołując w konsoli narzędzie `help()`, a następnie wpisując polecenie `keywords`, co przyniesie następujący efekt:



```
Python 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> help()

Welcome to Python 3.10's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.10/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

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

Oto przykład funkcji bezparametrowej, która nie zwraca żadnej wartości:

```
def hello(): #nagłówek
    print("Hello world") #pojedyncza instrukcja
```

`hello()` #wywołanie funkcji w dowolnym miejscu programu

Jak widać każdy zestaw instrukcji da się „obudować” w funkcję i wywołać ją w dowolnym miejscu. Aby uzyskać ten sam efekt, w tym wypadku wystarczyłoby przecież napisać: `print("Hello world")`. Jednak dzięki funkcji `hello()` możemy ten napis wywoływać wielokrotnie w programie, skracając tym samym kod. Trzeba również pamiętać, że na ogół funkcje liczą wiele (czasami kilkadziesiąt i więcej) linii, a wtedy oszczędność na kodzie (tym samym na pracy programisty) może być znacznie większa.

Często w kodach pythonowych zdarza się umieszczanie funkcji o nazwie `main()`, która wywoływana jest po uruchomieniu programu (na wzór języka C/C++) i w razie potrzeby sama wywołuje inne funkcje. Funkcja `main()` zawiera w takim wypadku tzw. logikę główną, której zadaniem jest przejrzyste przedstawienie zależności obowiązujących w kodzie programie. Zmodyfikujmy więc, w tym wypadku trochę sztucznie, poprzedni (bardzo prosty) kod i wyposażmy go w funkcję `main()`.

```
def main(): #definicja funkcji main()
    print("Teraz wygłoszę słynne słowa:")
    hello() #wywołanie funkcji hello()
```

```
def hello(): #definicja funkcji hello()
    print("Hello world")
```

`main()` #wywołanie funkcji main()

Wywołanie znajdującej się w ostatniej linijce kodu funkcji `main()` spowoduje przekazanie sterowania do definicji tej funkcji, z wnętrza której wywołana zostanie funkcja `hello()`. Dzięki temu na ekranie otrzymamy dwulinijkowy komunikat:

Teraz wygłoszę słynne słowa:

```
Hello world
```

Na razie nie widać specjalnego sensu stosowania funkcji `main()`. Pojawi się on wtedy, gdy w kodzie będzie więcej funkcji wywoływanych przez `main()`.

Zdefiniujmy i wywołajmy teraz funkcję, która coś zwraca. Niech to będzie funkcja mnożąca dwie liczby całkowite 3 i 4.

```
def iloczyn(a, b):
    return a * b
wynik = iloczyn(3, 4)
print(wynik)
```

Zadaniem instrukcji `return a * b` jest zwrócenie (podstawienie) iloczynu w miejscu wywołania funkcji wraz z podaniem liczb `a` i `b` (czyli argumentów przekazywanych do funkcji), tj. po prawej stronie znaku równości trzeciej linijki kodu. Możemy pominąć zmienną `wynik` oraz instrukcję przypisania `wynik = iloczyn(3, 4)` i ostatnie dwie linie kodu połączyć w jedną, tj. `print(iloczyn(3, 4))`.

Należy pamiętać, że liczba argumentów musi dokładnie odpowiadać liczbie parametrów, w przeciwnym wypadku interpreter Pythona może zgłosić błąd. Argumenty przypisywane są parametrom kolejno od lewej do prawej. Gdybyśmy chcieli natomiast zmienić kolejność przekazywanych argumentów, to wywołanie funkcji należy odpowiednio zmodyfikować, np. `print(iloczyn(b = 3, a = 4))`. W przypadku iloczynu nie ma to oczywiście znaczenia, ale w innych obliczeniach może mieć.

Ciekawą własnością języka Python jest też możliwość przypisania funkcji do zmiennej, a następnie potraktowanie tej zmiennej jako funkcji. Wynika to stąd, że każda funkcja w Pythonie jest obiektem, który tworzony jest w momencie interpretowania kodu, a zmienna tworzy w tym wypadku jedynie połączenie z obiektem, a nie jego kopię. W celu zobrazowania tej własności zmodyfikujmy ostatni kod.

```
def iloczyn(a, b):
    return a * b
wynik = iloczyn
print(wynik(3, 4))
```

Oczywiście, program powinien wchodzić w interakcję z użytkownikiem, więc zatrudnijmy też tego ostatniego w następnym przykładzie.

Przykład

Korzystając z funkcji napisz program, który dzieli przez siebie dwie liczby podane przez użytkownika. Zabezpiecz program przed dzieleniem przez zero.

```
def iloraz(a, b):
    if (b == 0):
        print("Działanie niedozwolone: b = 0")
    else:
        return a / b

a = int(input("Podaj a: "))
```

```
b = int(input("Podaj b: "))

print("iloraz = ", iloraz(a, b))
input("Aby zakończyć program wciśnij klawisz ENTER")
```

Funkcja może potrzebować własne zmienne do prawidłowego działania. W odróżnieniu od dostępnych w całym programie tzw. zmiennych globalnych, są to tzw. zmienne lokalne, które należą wyłącznie do tej funkcji i nie mogą być wywoływane poza nią. W wypadku, gdy inna funkcja zechce odwołać się do zmiennej lokalnej należącej do tamtej funkcji, Python wygeneruje błąd. Poniżej przykład takiego błędu.

```
def main():
    get_name()
    print(' Witaj, ', name)

def get_name():
    name = input('Podaj swoje imię: ')

main()
```

Zmienna `name` ma zasięg (ang. *scope*) mieszczący się wyłącznie w zakresie działania funkcji `get_name`, zatem instrukcje funkcji `main()` nie mają do niej dostępu. Błąd zasięgu pojawi się również, gdy instrukcja w funkcji odwołuje się do zmiennej, która pojawi się dopiero później.

Przyjrzyjmy się innemu przykładowi:

```
x = 2 #zmienna globalna x (widoczna w całym programie)
def example(y):
    x = y #zmienna lokalna x (widoczna tylko w funkcji example())
    print(x)

example(1)
print(x)
```

Obie zmienne `x` nie mają ze sobą nic wspólnego (poza nazwą). Dlatego na ekranie pojawią się dwie różne liczby, tj. 1 i 2, czyli wartość zmiennej globalnej nie ulegnie zmianie. Gdybyśmy chcieli zmienić wartość zmiennej globalnej `x`, moglibyśmy skorzystać ze słowa kluczowego `global`. W ten sposób program korzysta tylko z jednej zmiennej `x`, która widoczna jest wszędzie. Dzięki temu na ekranie pojawią się dwie liczby 1.

```
x = 2 #zmienna globalna x
def example(y):
    global x
    x = y #zmienna globalna x
    print(x)

example(1)
print(x)
```

Pytanie, co pojawi się na ekranie po wykonaniu poniższego programu, w którym druga funkcja wywoływana jest z wnętrza pierwszej?

```
def example():
    x = 2
```

```

print(x)
def example1():
    x = 3
    example1()
print(x)
example()

```

W tym wypadku pojawi się dwukrotnie liczba 2, ponieważ nastąpiło dwukrotne wywołanie pierwszej zmiennej lokalnej `x` w funkcji `print()` (druga zmienna `x` nie jest widziana). Możemy to zmienić stosując słowo kluczowe `nonlocal`, dzięki czemu na ekranie pojawią się kolejno liczby 2 i 3, ponieważ druga wartość `x` zacznie być widziana przez funkcję `example1()`.

```

def example():
    x = 2
    print(x)
    def example1():
        nonlocal x
        x = 3
    example1()
    print(x)
example()

```

Przykład

Napisz funkcję, której zadaniem jest zamiana wartości dwóch zmiennych.

```

def zamiana(a, b):
    temp = a
    a = b
    b = temp
    print("\nPozamianie a =", a, "b =", b)

a = int(input("Podaj wartość zmiennej a: "))
b = int(input("Podaj wartość zmiennej b: "))
print("\nPrzed zamianą a =", a, "b =", b)
zamiana(a, b)
input("\nNaciśnij dowolny klawisz...")

```

Przykład

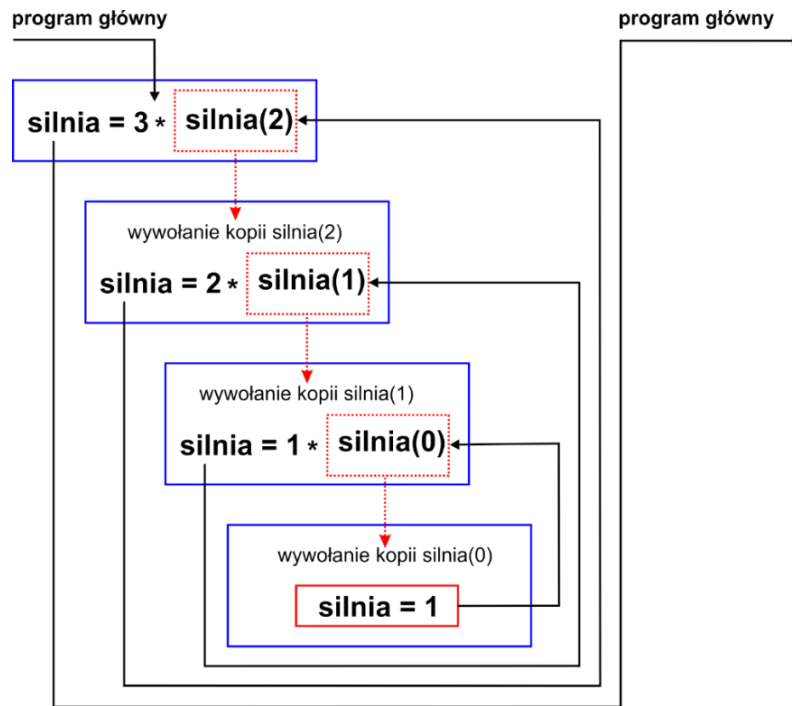
Za pomocą funkcji napisz program, który oblicza silnię liczby naturalnej w sposób rekurencyjny.

Uwaga:

Rekurencja (lub inaczej rekursja – ang. *recursion*) oznacza zdolność funkcji lub procedury do wywołania siebie samej. Rekurencyjna definicja rekurencyjna silni jest natomiast następująca:

$$n! = \begin{cases} 1 & \text{dla } n = 0 \\ n \cdot (n - 1)! & \text{dla } n > 0 \end{cases}$$

Działanie rekurencji dla 3! jest następujące:



```

n = int(input("Podaj n: "))
def silnia(n):
    if n == 0:
        return 1
    return n * silnia(n - 1)

print("Silnia liczby", n, "wynosi:", silnia(n))

```

Uwaga:

Python oferuje tzw. wyrażenie lambda (słowo kluczowe), zwane inaczej formą lambda, którego zadaniem jest wykonanie instrukcji w sposób podobny do funkcji. Mówimy, że lambda tworzy tzw. funkcję anonimową (to pojęcie znane jest też w innych językach programowania). Wyrażenie lambda stosowane jest często tam, gdzie z jakiegoś powodu nie można zastosować instrukcji def.

Przykład użycia:

''' definicja wyrażenia i zmiennych (parametrów), a po dwukropku wypisane są operacje, które należy przeprowadzić na tych zmiennych

```
x = lambda a, b, c: a + b + c
```

#wywołanie funkcji i przekazanie jej argumentów

```
print(x(1, 2, 3))
```

Powyższy przykład równoważny jest następującej funkcji:

```

def suma(a, b, c):
    return a + b + c
print(suma(1, 2, 3))

```

Ćwiczenia do samodzielnego rozwiązania, które należy wykonać wykorzystując pojęcie funkcji:

Ćwiczenie 1.

Napisz funkcję, której zadaniem jest wypisanie liczb od 9 do 100, które nie są podzielne przez 3 i 5.

Ćwiczenie 2.

Napisz rekurencyjnie cztery funkcje, które generują następujące ciągi n-elementowe (liczbę n podaje użytkownik):

- a) (3; 6; 9; 12; 15; 18; 21; 24...)
- b) (8; -4; 2; -1; 0,5; -0,25; 0,125...)
- c) (2; 6; 3; 7; 4; 8; 5; 9; 6; 10; 7; 11...)
- d) (1; 2; -4; -3; -1; -5; -8; -9; -14...)

Ćwiczenie 3.

Napisz funkcję, która sprawdza, czy dwie liczby naturalne są bliźniacze. Liczby bliźniacze są to takie dwie liczby pierwsze, których różnica wynosi 2, na przykład: 3 i 5, 5 i 7, 11 i 13. Wersja trudniejsza tego zadania polega na wyznaczeniu wszystkich liczb pierwszych z zakresu od 2 do 1000, które nie posiadają bliźniaka (np. liczba 37).

Temat: Listy, krotki i napisy

Listy w Pythonie zaliczane są do tzw. typów sekwencyjnych zmiennych, które podobne są do tablic w innych językach programowania, ale mają większe możliwości. Oznacza to, że lista jest kontenerem danych przechowującym obiekty (elementy) w ściśle określonej kolejności. Obiekty jednej listy mogą być różnych typów, a w szczególności mogą być też listami. Lista jest modyfikowalna (ang. *mutable* – mutowalna), czyli możemy zmieniać elementy listy w trakcie działania programu. Graficzną reprezentację listy i sposób indeksowania przedstawia rysunek poniżej.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
't'	'e'	'c'	'h'	'n'	'i'	'k'	' '	'p'	'r'	'o'	'g'	'r'	'a'	'm'	'i'	's'	't'	'a'

Listy można tworzyć w Pythonie na różne sposoby. Pustą listę można zapisać wykorzystując funkcję `list()`:

```
>>>liczby = list()
>>>liczby
[]
```

Można też utworzyć pustą listę używając po prostu nawiasów kwadratowych (później możemy dodawać do niej elementy):

```
>>>liczby = []
>>>liczby
[]
```

Tę samą funkcję `list()` możemy też wykorzystywać do konwersji innych typów danych na listę. W tabeli poniżej pokazane są przykłady operacji na listach.

Operacje na listach	Opis operacji
<code>lista = ["jeden", 2, 3.0, "cztery"]</code>	Przypisanie wartości listy zmiennej o nazwie <code>lista</code>
<code>lista[0]</code>	Odwołanie do pierwszego elementu, wynik: "jeden"
<code>lista[-1]</code>	Odwołanie do ostatniego elementu, wynik: "cztery"
<code>lista[-2]</code>	Odwołanie do przedostatniego elementu, wynik: 3.0
<code>lista[2:]</code>	Odwołanie do elementów począwszy, od elementu <code>lista[2]</code> , wynik: 3.0, "cztery"
<code>lista[1::2]</code>	Odwołanie do co drugiego elementu, począwszy od elementu <code>lista[1]</code> , wynik: 2, "cztery"
<code>lista *= 2</code>	Zdublowanie listy, wynik: "jeden", 2, 3.0, "cztery", "jeden", 2, 3.0, "cztery"
<code>lista[:2]</code>	Skrócenie listy, wynik: "jeden", 2
<code>len(lista)</code>	Długość (liczba elementów listy), wynik: 4
<code>lista[1] = 5</code>	Zmiana elementu o indeksie 1, wynik: "jeden", 5, 3.0, "cztery"
<code>lista[2] += 8</code>	Zwiększenie o 8 elementu o indeksie 2, wynik: "jeden", 5, 11.0, "cztery"

Poniżej zestawiono najpopularniejsze metody służące do przetwarzania list, przy czym kropkowy zapis np. `lista.append()` oznacza wywołanie metody `append` działającej na rzecz obiektu o nazwie `lista` (lub innego).

Metody przetwarzania list	Opis i wynik działania
<code>lista = [9, 2, 3, 4, 5, 6, 7, 8, 1, 10]</code>	
<code>lista.append(11)</code>	Dołączenie (dodanie) elementu do listy, wynik: [9, 2, 3, 4, 5, 6, 7, 8, 1, 10, 11]
<code>lista.extend([12, 13])</code>	Rozszerzenie listy o listę [12, 13], wynik: [9, 2, 3, 4, 5, 6, 7, 8, 1, 10, 12, 13]
<code>lista.count(8)</code>	Obliczenie, ile razy element o wartości 8 znajduje się na liście, wynik: 1
<code>lista.index(1)</code>	Indeks (pozycja) pierwszego wystąpienia elementu 1, wynik: 8
<code>lista.insert(4, 17)</code>	Wstawienie elementu o wartości 17 na pozycję (indeks) 4, wynik: [9, 2, 3, 4, 17, 5, 6, 7, 8, 1, 10]
<code>lista.pop(2)</code>	Zwraca i usuwa element na pozycji (indeksie) 2, wynik: [9, 2, 4, 5, 6, 7, 8, 1, 10]
<code>lista.remove(1)</code>	Usuwa z listy pierwszy znaleziony element o wartości 1, wynik: [9, 2, 3, 4, 5, 6, 7, 8, 10]
<code>lista.reverse()</code>	Odwraca kolejność elementów listy, wynik: [10, 1, 8, 7, 6, 5, 4, 3, 2, 9]
<code>lista.sort()</code>	Sortuje listę w kierunku niemalejącym, wynik: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Krotka (ang. *tuple*) jest listą (kontenerem), która nie może być modyfikowana (ang. *immutable* – niemutowalna). Zawartość krotki definiowana jest wyłącznie w trakcie tworzenia. Krotki stosujemy tam, gdzie zależy nam na niezmienności. Taki warunek spełniają np. dane dotyczące daty, miejsca urodzenia i PESEL pracownika albo lista współrzędnych geograficznych, które nie mogą ulegać zmianie. W związku z tym większość metod nie może być używana do działań na krotkach.

Do zapisywania krotek wykorzystywane są nawiasy okrągłe, zaś poszczególne elementy rozdzielane są przecinkami. Definicja krotki musi zawierać przecinek, nawet jeśli zawiera tylko jeden element np.: `krotka = (1,)`, ponieważ w ten sposób wiadomo, że to jest krotka, a nie ciąg znaków. Do definicji krotki nie są konieczne nawiasy, np.: `krotka = "rzeczownik", "czasownik", "przymiotnik"`, ale jeśli krotka ma być pusta, to nawiasy są konieczne, np.: `krotka = ()`.

Przykład 1.

Napisz funkcję, która oblicza sumę wszystkich elementów listy lub krotki.

```
def suma(lista):
    n = len(lista)
    s = 0
    for i in range(n):
        s += lista[i]
    return s

print("suma = ", suma([1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

```
def suma(krotka):
    n = len(krotka)
    s = 0
    for i in range(n):
        s += krotka[i]
    return s

print("suma = ", suma((1, 2, 3, 4, 5, 6, 7, 8, 9)))
```

Jak widać funkcja jest uniwersalna, ponieważ przyjmuje dane od listy i krotki, a w wyniku otrzymamy liczbę 45. Jedyna różnica dla listy i krotki tkwi w wywołaniu.

Przykład 2.

Napisz funkcję, która oblicza iloczyn wszystkich elementów listy mniejszych od 8.

```
def iloczyn(lista):
    n = len(lista)
    s = 1
    for i in range(n):
        if lista[i] < 8:
            s *= lista[i]
    return s

print("iloczyn = ", iloczyn([1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

W wyniku otrzymamy liczbę 5040.

Przykład 3.

Napisz funkcję, która sprawdza, czy na liście występują tylko elementy mające wartości dodatnie.

```
def check(lista):
    n = len(lista)
    for i in range(n):
        if lista[i] <= 0:
```

```

        return False
    return True

print(check([2, 3, 5, 0, 4]))

```

Wynik: False

Przykład 4.

Napisz funkcję, która wypisuje elementy listy dwuwymiarowej `[[2, 3, 4, 5], [7, 6, 4, 5], [8, 9, 4, 5]]` z podziałem na wiersze.

```

def wypisz(lista, m): #m - liczba elementów listy, które mają być wy-
    pisane
    for i in range(m):
        print(lista[i])

wypisz([[2, 3, 4, 5], [7, 6, 4, 5], [8, 9, 4, 5]], 3)

```

Wynik:

```

[2, 3, 4, 5]
[7, 6, 4, 5]
[8, 9, 4, 5]

```

Tę samą funkcję można też napisać w inny sposób i uzyskać dokładnie taki sam wynik:

```

def wypisz(lista):
    for i in lista:
        print(i)

wypisz([[2, 3, 4, 5], [7, 6, 4, 5], [8, 9, 4, 5]])

```

Przykład 5.

Napisz funkcję, która oblicza sumę tych elementów ww. listy dwuwymiarowej, które nie są podzielne przez 3.

```

def oblicz(lista, m, n):
    s = 0
    for i in range(m):
        for j in range(n):
            if lista[i][j] % 3:
                s += lista[i][j]
    return s

print(oblicz([[2, 3, 4, 5], [7, 6, 4, 5], [8, 9, 4, 5]], 3, 4))

```

albo inaczej

```

def oblicz(lista):
    s = 0
    for i in lista:
        for j in i:
            if j % 3:
                s += j

```

```

    return s

print(oblicz([[2, 3, 4, 5], [7, 6, 4, 5], [8, 9, 4, 5]]))

```

Listy w Pythonie można deklarować jeszcze inaczej. I tak listę jednowymiarową możemy zadeklarować i wypełnić od razu liczbami za pomocą konstrukcji: `lista = [0 for row in range(10)]`, która spowoduje (przykładowo) powstanie 10-elementowej listy wypełnionej samymi zerami. Podobnie możemy utworzyć (przykładowo) wyzerowaną listę dwuwymiarową 3×4 (trzy wiersze i cztery kolumny) za pomocą konstrukcji: `lista = [[0 for col in range(4)] for row in range(3)]`.

Przykład 6.

Napisz program, który generuje tablicę o wymiarach 4×3 (cztery wiersze i trzy kolumny), wypełnioną kolejno liczbami: 1, 3, 5, 6, 8, 10, 11, 13, 15, 16, 18, 20. Podane liczby powstają ze wzoru: $5i + 2j + 1$, gdzie i oraz j są indeksami odpowiednio wierszy i kolumn (indeksacja zaczyna się od zera).

```

def table(m, n):
    lista = [[0 for col in range(n)] for row in range(m)]
    for i in range(m):
        for j in range(n):
            lista[i][j] = i * 5 + j * 2 + 1
        print(lista[i])

table(4, 3)

```

Napisy, zwane również łańcuchami znakowymi, są niezmiennie. Jest to więc typ sekwencyjny niezmienny, zatem nie jest możliwe przypisywanie wartości pojedynczym elementom napisu. Możliwe są za to inne operacje. Wartości napisów podajemy w cudzysłowach lub apostrofach. Właściwość ta przydaje się, gdy chcemy użyć wewnątrz napisu cudzysłów lub apostrof, przykładowo instrukcja: `print("Zespół Szkół 'Elektryk' w Słupsku")` wygeneruje błąd, którego możemy uniknąć stosując w miejsce cudzysłowu wewnętrzny apostrof (lub odwrotnie), czyli `print("Zespół Szkół 'Elektryk' w Słupsku")`. Można też sięgnąć do znaków specjalnych i zastosować wsteczny ukośnik (ang. *backslash*): `print("Zespół Szkół \"Elektryk\" w Słupsku")`. Do napisów stosuje się wiele operatorów wymienionych przy okazji list i krotek. W tabeli poniżej znajduje się kilka innych.

Operacje na napisach	Opis operacji
<code>tekst = "programista"</code> <code>tekst = 'programista'</code>	Przypisanie wartości napisu zmiennej o nazwie tekst
<code>tekst = 'technik '</code> <code>+ tekst</code>	Konkatenacja (dodawanie, złączenie napisów), wynik: 'technik programista'
<code>len(tekst)</code>	Oblicza długość napisu (programista), wynik: 11
<code>chr(65)</code>	Zwraca znak o kodzie ASCII równym dziesiętnie 65, wynik: 'A'
<code>ord(tekst[0])</code>	Zwraca kod ASCII znaku tekst[0], czyli tutaj litery 'p', wynik: 112
<code>==</code> <code>!=</code> <code><</code> <code>></code>	Operatory relacji (napisy można porównywać)
<code>'p' in tekst</code> <code>"ro" not in tekst</code>	Sprawdza, czy w napisie znajduje się podany znak lub znaki

Podobnie, jak w przypadku list i krotek, obiekty łańcuchowe korzystają z różnych metod.

Metody przetwarzania napisów	Opis i wynik działania
tekst = "Programista to bardzo dobry zawód"	
<code>tekst.count('a')</code>	Oblicza liczbę znaków 'a' w napisie, wynik: 4
<code>tekst.find('to')</code>	Znajduje pierwsze wystąpienie ciągu znaków 'to' w napisie, wynik: 12. Jeśli nie znajdzie, zwraca wynik -1
<code>tekst.rfind('to')</code>	Znajduje ostatnie (pierwsze od końca) wystąpienie ciągu znaków 'to' w napisie, wynik: 12. Jeśli nie znajdzie, zwraca wynik -1
<code>tekst.isdigit()</code> <code>tekst.isalpha()</code> <code>tekst.isalnum()</code>	Zwraca True, jeśli w napisie są odpowiednio: same cyfry, same litery albo same litery i cyfry, wynik (w każdym przypadku): False (bo spacja jest symbolem specjalnym)
<code>tekst.lower()</code>	Zamienia litery na małe, wynik: 'programista to bardzo dobry zawód'
<code>tekst.upper()</code>	Zamienia litery na wielkie, wynik: 'PROGRAMISTA TO BARDZO DOBRY ZAWÓD'
<code>tekst.swapcase()</code>	Odwraca wielkość liter, wynik: 'pROGRAMISTA TO BARDZO DOBRY ZAWÓD'
<code>tekst.capitalize()</code>	Zamienia pierwszą literę na wielką, a pozostałe na małe, wynik: (tutaj bez zmian)
<code>tekst.strip()</code> <code>tekst.lstrip()</code> <code>tekst.rstrip()</code>	Usuwa zbędne tzw. białe znaki (spacje, tabulatory, znaki nowego wiersza) odpowiednio: z lewej i prawej strony napisu, tylko z lewej strony napisu, tylko z prawej strony napisu, wynik: (tutaj bez zmian, bo nie ma takich znaków)
<code>tekst.replace('zawód', 'fach')</code>	Zamienia wszystkie wystąpienia łańcucha 'zawód' na 'fach', wynik: 'Programista to bardzo dobry fach'
<code>tekst.split()</code>	Konwertuje napis na listę wyrazów, wynik: ['Programista', 'to', 'bardzo', 'dobry', 'zawód']

Przykład 7.

Napisz program, który sprawdza, czy wyraz podany przez użytkownika jest palindromem (tak samo czyta się wprost i wspak, na przykład kajak).

```
def palindrom(s):
    n = len(s)
    for i in range(n // 2):
        if s[i] != s[n - i - 1]:
            return False
    return True

wyraz = input("Podaj wyraz: ")

if palindrom(wyraz):
    print("Wyraz", wyraz, "jest palindromem")
else:
    print("Wyraz", wyraz, "nie jest palindromem")
```

Zadania do samodzielnego rozwiązania

Zadanie 1.

Masz listę składającą się z liczb całkowitych: [9, 4, 3, 5, 7, 9, 4, 2, 1, 5]. Napisz jeden program, który w formie krótkich funkcji:

- a) wypisuje wszystkie elementy listy,
- b) oblicza sumę tych elementów listy, których indeks nie jest podzielny przez 5,
- c) zwiększa o 2 te elementy listy, które mają nieparzysty indeks zawarty w przedziale [3, 8] i wypisuje tę listę,
- d) sprawdza, czy wartości wszystkich elementów na liście są nieujemne,
- e) oblicza iloczyn tych elementów listy, których wartość równa jest 5,
- f) sprawdza, czy na liście znajduje się element, którego wartość nie zawiera się w przedziale [5, 8].

Zadanie 2.

Masz napis: `informatyka`. Napisz jeden program, który w formie krótkich funkcji:

- a) wypisuje znaki różne od 'm' i 'k',
- b) oblicza liczbę znaków 'a',
- c) oblicza liczbę znaków różnych od 'f', które mają indeks parzysty,
- d) zamienia znak 'o' na 'x',
- e) zamienia znaki różne od 'm' i 'a' na 'w'.

Zadanie 3.

Palindromami mogą być również całe zdania, na przykład:

Oko w oko
Nogawka jak wagon
Kobyła ma mały bok
Może jutro ta dama sama da tortu jeżom

Napisz program, który sprawdzi, czy powyższe zdania (lub inne) są palindromami. Algorytm nie powinien uwzględniać spacji oraz wielkości liter.

Zadanie 4.

Napisz program sprawdzający, czy dana liczba naturalna jest „podzielna”, tj. jest większa od zera i dzieli się przez sumę swoich cyfr (taką liczbą jest np. 21). Ponadto – zgodnie z wyborem użytkownika – program powinien generować wszystkie liczby podzielne do 10000.