Sfinansowano ze środków Narodowego Funduszu Ochrony Środowiska i Gospodarki Wodnej

Ministerstwo Klimatu i Środowiska

OnGeo

Programowanie w języku Python – składnia

MATERIAŁY SZKOLENIOWE

Spis treści

1. Wprowadzenie ... 3
2. Python – podstawy ... 4
2.1. Składnia ... 4
WCIĘCIA KODU ... 5
SŁOWA ZAREZERWOWANE ... 5
KOMETARZE ... 6
OPERATORY ... 6
2.2. Zmienne i typy danych ... 8
WBUDOWANE TYPY DANYCH ... 9
ZMIENNE ... 11
2.3. Pętle i wyrażenia warunkowe ... 13
WYRAZENIA WARUNKOWE IF ... 13
PĘTLA WHILE ... 15
PĘTLA FOR ... 15
INSTRUCKE BREAK I CONTINUE ORAZ KLAUZULA ELESE W PĘTLACH ... 17
2.4. Praca z wbudowanymi typami danych ... 19
LICZBY (NUMBERS) ... 19
TEKSTY (STRINGS) ... 22
LISTY (LISTS) ... 31
KROTKI (TUPLES) ... 36
SŁOWNIKI (DICTIONARIES) ... 37
2.5. Operacje na plikach ... 40
2.6. Błędy oraz obsługa błędów ... 44
3. Funkcje ... 48
4. Klasy ... 50

1. Wprowadzenie

W materiałach szkoleniowych wykorzystane zostały materiały udostępnione na stronie https://www.ekoportal.gov.pl/.

Przy opisywaniu funkcji języka Python zastosowano sposób opisu parametrów zgodny z oficjalną dokumentacją (http://www.python.org/doc/). Nazwy parametrów podawane są w języku angielskim, ale zawsze objaśniane są w opisie działania funkcji. Parametry obligatoryjne podawane są w nawiasach kwadratowych.

W celu ułatwienia pracy z materiałami, na szarym tle (przykład poniżej) umieszczono nazwy plików z zawierających prezentowane skrypty lub rozwiązania zadań.

Przykładowy_skrypt.py

2. Python – podstawy

Python jest jednym z wielu dostępnych obecnie języków programowania. Z punktu widzenia przetwarzania danych przestrzennych, spośród innych języków, Python wyróżnia się bogatym zestawem bibliotek dedykowanych do działań na danych o określonej lokalizacji w przestrzeni.

Python jest obiektowo - zorientowanym językiem skryptowym. Należy też do grupy języków interpretowanych, co oznacza, że na komputerze, na którym działamy musi znajdować się interpreter, który potrafi odczytać i wykonać kod źródłowy zapisany w Pythonie.

Do najważniejszych cech Pythona upraszczających pisanie w nim skryptów należą:

- dynamiczna kontrola typów danych,
- prosta i przejrzysta składnia,
- wspomniany wcześniej bogaty zestaw gotowych bibliotek.

Zwięzły opis idei programowania w języku Python zawarty jest w jednym z jego modułów. Pojawia się on od razu po zaimportowaniu modułu **this**.

```
>>> import this
```

2.1. Składnia

Python jest językiem programowania, w którym duży nacisk kładziony jest na klarowność i przejrzystość kodu. Składania Pythona projektowana była z myślą o zapewnieniu jak największej czytelności i zwięzłości zapisanych za jej pomocą programów. W odróżnieniu od wielu języków programowania, znaczenie ma wielkość liter i wcięcia poszczególnych linii kodu. Puste linie, spacje i komentarze są pomijane. Wyrażenia wykonywane są po kolei, chyba że zdefiniowano inaczej.

WCIĘCIA KODU

W języku Python wcięcia kodu wykorzystywane są do oznaczenia bloków kodu, które na ich podstawie są automatycznie wykrywane. Liczba spacji we wcięciach jest dowolna, przy założeniu, że jest ona stała dla jednego bloku. Poniżej przykłady poprawnie i niepoprawnie wciętych bloków:

Poprawne dwa bloki kodu:

```
if True:
    print("Pierwszy wiersz pierwszego bloku")
    print("Drugi wiersz pierwszego bloku")
else:
    print("Drugi blok")
```

Niepoprawnie wcięty pierwszy blok:

```
if True:
print("Pierwszy wiersz pierwszego bloku")
print("Drugi wiersze pierwszego bloku")
else:
  print("Drugi blok")
```

SŁOWA ZAREZERWOWANE

Nazwy zmiennych, funkcji itd. mogą zawierać cyfry, litery i podkreślenia, ale nie mogą zaczynać się od cyfr. Wielkość liter w nazwach ma znaczenie. W charakterze nazw nie mogą być także wykorzystywane słowa zarezerwowane, których listę można wywołać po zaimportowaniu modułu **keyword**.

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

KOMETARZE

Komentarze oznaczane są znakiem '**#**' na początku linii. Wszystko, co w danej linii znajduje się po znaku '#' jest traktowane, jako komentarz.

```
>>> # Cała linia może być komentarzem
>>> x = 2 # Fragment linii może być komentarzem
```

W Pythonie nie występuje znak kończący komentarz. Komentarz kończy się wraz z końcem linii.

OPERATORY

W tabelach poniżej zestawiono operatory, które mogą być wykorzystane w skryptach pisanych w języku Python.

Operatory arytmetyczne:

Operator	**Opis**	**Przykład**	
		a = 5.0, b = 3.0	c
+	Dodawanie	c = a + b	8.0
-	Odejmowanie lub zmiana znaku liczby (oprócz 0)	c = a – b	2.0
*	Mnożenie	c = a * b	15.0
/	Dzielenie	c = a / b	1.6666666666666667
%	Modulo (reszta z dzielenia)	c = a % b	2.0
**	Potęgowanie	c = a ** b	125.0
//	Dzielenie z odrzuceniem części dziesiętnej	c = a // b	1.0

Operatory porównania:

Operator	**Opis**	**Przykład**	
		a = 5.0, b = 3.0	wynik
==	Operator równości	a==b	False
!=	Różny od	a!=b	True
<>	Różny od	a<>b	True
>	Większy	a>b	True
<	Mniejszy	a<b	False
>=	Większy lub równy	a>=b	True
<=	Mniejszy lub równy	a<=b	True

Operatory przypisania:

Operator	**Opis**	**Przykład**	
		składnia	odpowiednik
=	Prosty operator przypisania	c = a + b	c = a + b
+=	Dodawanie i operator przypisania	c += a	c = c + a
-=	Odejmowanie i operator przypisania	c -= a	c = c - a
*=	Mnożenie i operator przypisania	c *= a	c = c * a
/=	Dzielenie i operator przypisania	c /= a	c =c / a
%=	Modulo i operator przypisania	c %= a	c = c % a
**=	Potęgowanie i operator przypisania	c ** a	c = c ** a

Operatory logiczne:

Operator	Opis	Przykład	
		składnia	odpowiednik
and	Operator logiczny ‘i’	a and b	False
or	Operator logiczny ‘lub’	a or b	True
		b or a	True
not	Operator logiczny ‘zaprzeczenie’	not(a and b)	True

Operatory przynależności:

Operator	Opis	Przykład	
		a =5.0, b=3.0	wynik
in	Zawiera się w	a in [5.0 , 3.0]	True
not in	Nie zawiera się w	b not in [5.0, 3.0]	False

Operatory identyczności:

Operator	Opis	Przykład	
		a = 5.0, b = 3.0	wynik
is	Jest identyczny	a is b	False
is not	Nie jest identyczny	a is not b	True

2.2. Zmienne i typy danych

Pod pojęciem typu danych w programowaniu rozumiany jest rodzaj, struktura i zakres wartości jakie przyjmować może np. literał, zmienna, argument funkcji. W języku Python wyróżnić można grupy typów danych sekwencyjnych i niesekwencyjnych oraz typów danych zmiennych (ang. mutable) i niezmiennych (ang. Immutable).

WBUDOWANE TYPY DANYCH

W języku Python korzystać można z następujących typów danych:

Kategoria	Nazwa typu	Opis
	NoneType	Pusta wartość (Null)
Liczbowe	int	Liczby całkowite z zakresu -2147483648 do 2147483647*
	long	Liczby całkowite długie
	float	Liczby zmiennoprzecinkowe. Założenia precyzji są uzależnione od platformy, na jakiej działa interpreter.
	complex	Liczby zespolone
	bool	Wartości logiczne – True lub False
Sekwencyjne	str	Łańcuchy znaków
	unicode	Łańcuchy znaków Unicode
	tuple	Krotka
	list	Listy
	bytearray	Tablice
	xrange	Sekwencje
Zbiory	set	Zbiory zmienne
	frozenset	Zbiory niezmienne
Odwzorowania	dict	Słowniki
Pliki	file	Pliki

Typ danych sprawdzić można wywołując funkcję **type().**

type(*object***)** –zwraca typ podanego obiektu

object – dowolny obiekt języka Python

```
>>> type(214748364755)
<type 'long'>

>>> type(12)
<type 'int'>

>>> type("Ala")
<type 'str'>

>>> type(True)
<type 'bool'>

>>> type([1,2,3])
<type 'list'>

>>> type((1,2,3))
<type 'tuple'>
```

Sekwencyjność danego typu wiąże się z możliwością przechowywania uporządkowanych ciągów wartości w jednej zmiennej. Przykładem może być typ danych tekstowych gdzie napisy są sekwencjami poszczególnych znaków. Operacje na tekstach możemy wykonywać odwołując się do całego napisu, jego fragmentu lub do poszczególnych znaków.

W przypadku typów niesekwencyjnych (np. typów liczbowych), dana wartość jest zawsze traktowana jako całość – liczbę zawsze rozpatrujemy jako jedną wartość, nie można się odwołać do poszczególnych cyfr.

Do sekwencyjnych typów danych należą: teksty (str, unicode), listy (list), krotki (tuple).

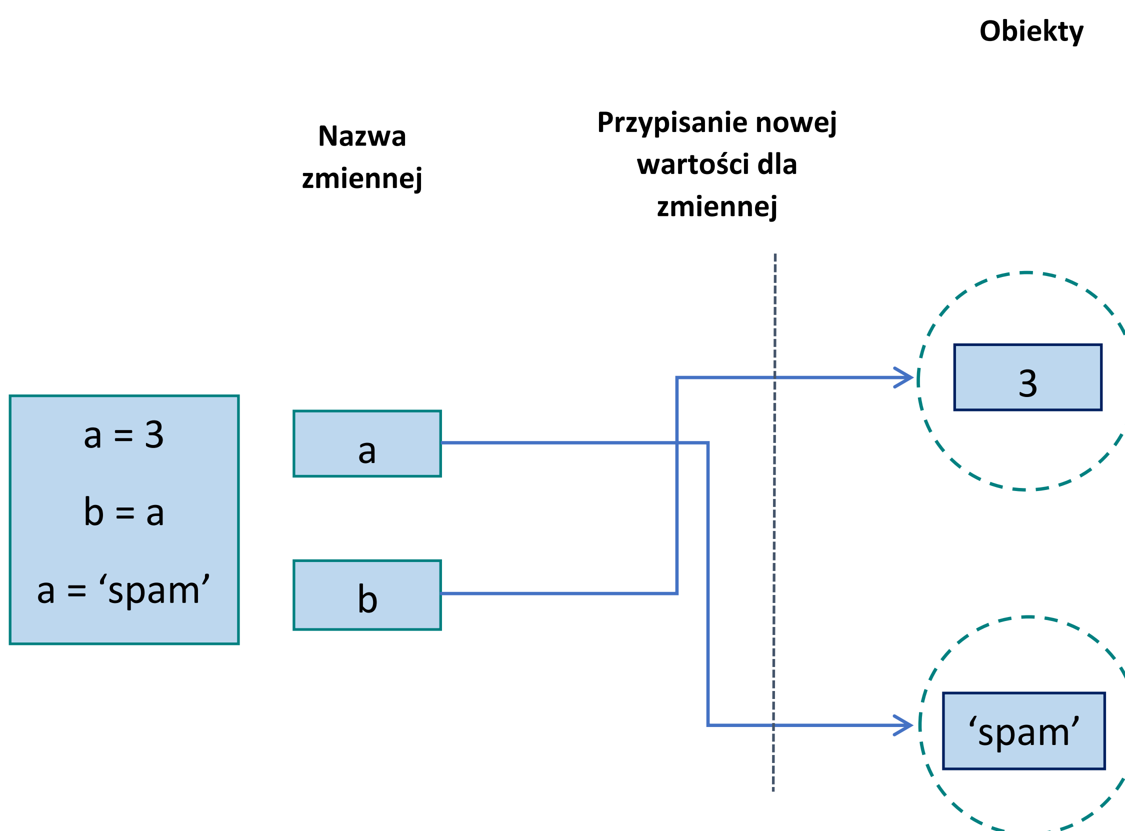
Drugi istotny podział uwzględnia rozróżnienie typów danych zmiennych (*ang. mutable*) – np. listy (list), zestawy (set), słowniki (dictionaries) oraz niezmiennych (*ang. inmutable*) – np. teksty (str), liczby (int, float, complex), krotki (tuple).

W pierwszym przypadku, modyfikować można cały obiekt lub jego elementy – np. można dodać, zmienić, usunąć element listy lub słownika. W przypadku typów niezmiennych, nie ma możliwości zmiany raz utworzonego obiektu – np. jeżeli dodamy jakąś wartość do obiektu typu liczbowego, tworzony jest zawsze nowy obiekt odpowiadający wynikowi sumy.

Sfinansowano ze środków Narodowego Funduszu Ochrony Środowiska i Gospodarki Wodnej
Ministerstwo Klimatu i Środowiska
OnGeo

ZMIENNE

Zmienne porównać można do „fragmentów” pamięci komputera, w których przechowywane mają być konkretne informacje. Zmienne w Pythonie nie mają typów. Typ danych przypisany jest dopiero do wartości zmiennej. Przykładowo ta sama zmienna może wskazywać zarówno obiekty tekstowe jak i liczbowe. Na rysunku poniżej przedstawiony jest schemat zmiany wartości zmiennej. Zmiennym o tych samych nazwach nadawane są nowe wartości. Odwołując się do zaktualizowanych zmiennych należy pamiętać, że nie przechowują one już informacji o swojej poprzedniej wartości.




```
>>> x = "Ala"
>>> type(x)
<type 'str'>
>>> x = 1
>>> type(x)
<type 'int'>
>>> x
1
```

Znak równości ('=') traktowany jest jako operator przypisania, za pomocą którego do zmiennej przypisywany jest konkretny obiekt.

Ta sama wartość może być przypisana do kilku zmiennych.

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

Do dowolnej zmiennej można przypisać wartość dowolnego typu, ale każda wartość typ danych ma przypisany na stałe.

Zmienne usuwane są za pomocą instrukcji **del**. Po usunięciu zmiennej, w przypadku próby wywołania przypisanej do niej wartości zwracany jest błąd.

```
>>> del(x)
>>> x
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    x
NameError: name 'x' is not defined
```

Ostatnia wartość wyświetlona w interpreterze przechowywana jest pod zmienną '_' i można ją wykorzystywać w dalszych obliczeniach.

```
>>> x = 2
>>> x + 4
6
>>> _ + 6
12
```

Ćwiczenie 1.

Stwórz zmienne: *wysokosc*, *podstawa*. Przypisz im dowolne wartości liczbowe. Na podstawie utworzonych zmiennych oblicz pole trójkąta dla wprowadzonych parametrów. Wynik zapisz w kolejnej zmiennej – *pole*.

pole_trojkata.py

```
wysokosc = 10
podstawa = 15

pole = 0.5*wysokosc*podstawa

print(pole)
```

Ćwiczenie 2.

Napisz program, który poprosi użytkownika o podanie wzrostu oraz wagi, i na tej podstawie obliczy współczynnik BMI (waga / wzrost^2).

```
waga = input("Ile ważysz [kg]: ")
wzrost = input("Ile masz wzrostu [m]: ")
wz = wzrost.replace(",",".")
BMI = float(waga) / (float((wz))**2)
print(BMI)
```

BMI.py

2.3. Pętle i wyrażenia warunkowe

Często zdarza się, potrzeba wielokrotnego powtórzenia tej samej części algorytmu dla różnych wartości. W tym celu wykorzystywane są tak zwane pętle. Jeżeli zachodzi potrzeba zastosowania różnych algorytmów w zależności od scenariusza, stosowane są wyrażenia warunkowe. Obie konstrukcje i ich różne wersje omówiono poniżej.

WYRAZENIA WARUNKOWE IF

Jeżeli warunek jest spełniony, blok kodu wewnątrz instrukcji warunkowej zostanie wykonany. Jeżeli warunek nie jest spełniony, blok kodu wewnątrz instrukcji warunkowej zostanie pominięty.

```
>>>  if x < 5:
      print("x jest mniejszy od 5")
```

Jeżeli warunek zapisany jest tylko w jednej linii, można umieścić go w tej samej linii co nagłówek:

```
>>> if x < 5: print("x jest mniejszy od 5")
```

Instrukcje warunkowe można zapisywać też w formie skróconej, jako wyrażenia warunkowe:

```
print("x jest mniejsze od 5" if x<5 else "x jest większe od 5")
```

Wewnątrz instrukcji warunkowej można umieszczać dowolną liczbę warunków:

```
>>>  x = 4
>>> y = 8
if x < y:
    print("y jest większe")
elif x > y:
    print("x jest większe")
else:
    print("x jest równy y")
```

Wyrażenia warunkowe można zagnieżdżać:

```
if x < y:
    print("y jest większe")
    if y > 10:
        print("y jest większe od 10")
    elif y == 10:
        print("y jest równe 10")
    else:
        print("y jest mniejsze od 10")
```

```
elif x > y:
    print("x jest większe")
    if x > 10:
        print("x jest większe od 10")
    elif x == 10:
        print("x jest równe 10")
    else:
        print("x jest mniejsze od 10")
else:
    print("x jest równy y")
```

PĘTLA WHILE

Składnia pętli warunkowej **while** wygląda następująco:

```
>>> x = 1
while x < 10:
    print("to jest obrót pętli nr: " + str(x))
     x += 1
```

Warunek sprawdzany jest przed każdą kolejną iteracją pętli. Gdyby w powyższym kodzie nie było zapisane, że po każdy obrocie pętli, x ma być zwiększany o zadaną wartość (x += 1), pętla „kręciłaby się" w nieskończoność.

PĘTLA FOR

Pętla **for** umożliwia iteracje ze wykorzystaniem wszystkich typów sekwencyjnych języka Python. Działa między innymi na ciągach tekstowych, listach i krotkach. Poniżej przykład z wykorzystaniem podstawowej składni:

```
>>>  word = "Python"
>>>  for letter in word:
      print(letter)
```

```
P
y
t
h
o
n
```

Jedną z funkcji często wykorzystywanych w połączniu w pętlą **for** jest funkcja **range()**, która generuje listę kolejnych liczb o zadanym przyroście arytmetycznym.

range(*[start,]stop,[step]***)** – generuje listę liczb w zadanym zakresie i o zadanym przyroście

start – liczba początkowa przedziału wartości listy

stop – liczba końcowa przedziału listy

step – przyrost arytmetyczny

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

>>> range(2,8)
[2, 3, 4, 5, 6, 7]

>>> range(0,10,2)
[0, 2, 4, 6, 8]
```

W celu ponumerowania poszczególnych liter w słowie „Python", wykorzystać można następującą pętlę:

```
>>> word = "Python"
>>> for letter in range(len(word)):
      print(letter, word[letter])

0 P
1 y
2 t
3 h
4 o
5 n
```

INSTRUCKE BREAK I CONTINUE ORAZ KLAUZULA ELESE W PĘTLACH

Zdarza się, że do poprawnego działania skryptu konieczne jest uwzględnienie możliwości przerwania pętli w określonych przypadkach. Pętle przerywać można na dwa sposoby – kończąc natychmiast działanie całego bloku kodu pętli, lub przerywając tylko określoną iterację.

break – powoduje natychmiastowe wyjście z pętli

continue – powoduje przerwanie bieżącej iteracji i przejście do kolejnej

```
>>> for letter in "Python in GIS":
      print(letter)
      if letter == " ":
            break
P
y
t
h
o
n
```

```
>>>   for letter in "Python in GIS":
            if letter in "Python in ":
                  continue
            print(letter)
G
I
S
```

W pętlach **for**, kod w klauzuli **else** wykonywany jest po wyczerpaniu wszystkich iteracji pętli. W pętlach **while**, kod ten wykonywany, gdy warunek pętli przestaje być spełniony.

Ćwiczenie 3.

Wymyśl algorytm, a następnie napisz program, który sumuje liczby dodatnie podawane przez użytkownika:

- Niech program prosi o podawanie kolejnych liczb dopóki użytkownik nie wprowadzi liczby ujemnej.
- Niech po każdej podanej liczbie program wypisuje aktualną wartość sumy.
- Na koniec niech program wypisuje całkowitą uzyskaną sumę.

Uwaga! Do rozwiązania zadania potrzebna jest funkcja która przyjmuje wprowadzoną przez użytkownika wartość i przekazuje do skryptu. Wykorzystaj jedną z funkcji opisanych poniżej:

input(*[prompt]*) – funkcja wyświetla na ekranie tekst przekazany w parametrze *prompt* i przekazuje do skryptu wprowadzoną przez użytkownika wartość.
input*([prompt])* **–** działa analogicznie do **input()** ale wprowadzona przez użytkownika wartość jest zawsze konwertowana na tekst.

```
x = 0
y = 0

while y >= 0:
    y = float(input("Podaj liczbę: "))
    print(y)
    x += y
    print("Aktualna wartość sumy = ", x)

print("Suma podanych liczb = ", x)
```

suma_petla.py

Ćwiczenie 4.

Napisz skrypt, który prosi użytkownika o podanie 10 różnych liczb podzielnych przez 3, a następnie wyświetla te liczby od najmniejszej do największej wraz z informacją czy dana liczba jest jednocyfrowa, dwucyfrowa, trzycyfrowa lub posiada większą liczbę cyfr.

Uwaga! Do rozwiązania tego zadania potrzebna jest znajomość typów danych List i String, które są omówione w następnym rozdziale.

podzielne_przez3.py

2.4. Praca z wbudowanymi typami danych

LICZBY (NUMBERS)

Najczęściej wykorzystywane typy liczbowe Pythona to liczby całkowite (integer) i zmiennoprzecinkowe (float). W zależności od tego, z jakiego typu korzystamy inne będą wyniki poszczególnych operacji matematycznych.

int:

```
>>> 3 / 2
1
```

float:

```
>>> 3.0 / 2.0
1.5
```

W przypadku mieszania typów liczbowych w działaniach arytmetycznych, typ prostszy jest przed wykonaniem działania konwertowany do typu bardziej złożonego. Hierarchia typów liczbowych pod względem złożoności w Pythonie wygląda następująco: liczby całkowite są prostsze od zmiennoprzecinkowych, liczby zmiennoprzecinkowe są prostsze od liczb zespolonych.

int / float:

```
>>> 3 / 2.0
1.5
```

Każdą liczbę można przekonwertować do innego typu liczbowego za pomocą odpowiednich funkcji:

int(*[x]***)** – konwertuje *x* na liczbę całkowitą lub zwraca ‘0’, jeżeli żaden argument nie został podany

```
>>> int(5.3444)
5
```

float(*[x]***)** – konwertuje *x* na liczbę zmiennoprzecinkową lub zwraca '0.0', jeżeli żaden parametr nie został podany

```
>>> float(5)
5.0
```

complex(*real [,imag]***)** – konwertują *na liczbę zespoloną*

real – *część rzeczywista liczby zespolonej*
imag – *część urojona liczby zespolonej*

```
>>> complex(5)
(5+0j)
```

Po zaimportowaniu modułu **math** korzystać można z wbudowanych funkcji matematycznych, w tym funkcji trygonometrycznych. Poniżej znajduje się zestawienie przykładowych funkcji modułu **math**. Pełną listę funkcji wraz z opisem można znaleźć w dokumentacji języka Python (http://docs.python.org/2/library/math.html).

Funkcja	Opis
math.fabs(x)	Wartość bezwzględna x
math.pow(x,y)	X podniesione do potęgi y
math.sqrt(x)	Pierwiastek kwadratowy z x
math.exp(x)	Exponent liczby x
math.pi	Stała π
math.e	Stałą e
cmp(x,y)	-1 jeżeli x < y, 0 jeżeli x = y, 1 jeżeli x > y
max(x1,x2…)	Największy z podanych argumentów
min(x1,x2…)	Najmniejszy z podanych argumentów
round(x,[n])	X zaokrąglony do n miejsc po przecinku

Jednym z ciekawszych modułów Python jest moduł **random**, który wykorzystywany jest do operacji opartych na pseudolosowości.

random.random() – zawraca losową liczbę zmiennoprzecinkową z zakresu 0.0 -1.0.

random.randint(*a,b*) – zwraca losową liczbę całkowitą (N) z podanego zakresu, spełniającą warunek a <= N <= b.

random.choice(*seq*) – zwraca losowy element z podanej sekwencji (np. listy), jeżeli podana sekwencja jest pusta zwracany jest błąd.

```
>>> random.choice('abcdefghijkl')
'd'
```

random.shuffle(*seq*) – przestawia losowo kolejność elementów w sekwencji

```
>>> lista = ["Ala", "Agnieszka", "Marcin", "Radek", "Monika", "Przemek", "Agata"]
>>> random.shuffle(lista)
>>> print(lista)
['Przemek', 'Marcin', 'Ala', 'Monika', 'Radek', 'Agnieszka', 'Agata']
```

Ćwiczenie 5.

Napisz skrypt, który poprosi użytkownika o podanie liczby, a następnie sprawdzi czy jest to liczba parzysta i wyświetli odpowiedni komunikat.

parzyste.py

Ćwiczenie 6.

Napisz skrypt, który prosi użytkownika o podanie trzech długości odcinków, a następnie sprawdza czy z wprowadzonych odcinków można zbudować trójkąt, wyświetla odpowiedni komunikat. Jeśli trójkąt może być zbudowany z podanych odcinków, skrypt sprawdza, jaki jest to trójkąt (prostokątny, ostrokątny czy rozwartokątny) i wyświetla odpowiedni komunikat.

Uwaga! Do rozwiązania tego zadania potrzebna jest znajomość List, które są omówione w następnym rozdziale.

trójkąt.py

TEKSTY (STRINGS)

Ciągi tekstowe tworzymy ograniczając dany łańcuch znaków cudzysłowami (” ”) lub apostrofami (‘ ’).

```
>>> 'Przykładowy tekst'
>>> "Inny przykładowy tekst"
>>> "Przykładowy tekst z 'apostrofem' w środku"
```

Jako ogranicznik łańcuchów znaków może być wykorzystywany także potrójny cudzysłów (””” ”””) lub potrójny apostrof (‘’’ ‘’’). Tak ograniczony ciąg znakowy może zawierać cudzysłowy, apostrofy, tabulatory itd. traktowane także jako tekst. W ciągu znakowym ograniczonym potrójnym cudzysłowem, uwzględnić można podział na linie.

```
>>> """To jest bardzo długi tekst
podzielony na kilka linii, zawierający cudzysłowy ("")
i apostrofy ('') w środku"""
```

Aby przekonwertować obiekt innego typu do typu tekstowego wykorzystywana jest funkcja **str()**.

str(*object***)** – zwraca reprezentację obiektu w postaci łańcucha znaków, jeżeli żaden parametr nie został podany –zwraca pusty ciąg znakowy

object – dowolny obiekt języka Python

```
>>> str(12345)
'12345'
```

Długość ciągu znakowego sprawdzana jest za pomocą funkcji **len()**.

len(*s***)** - zwraca długość (ilość elementów) obiektu *s*, zwykle wykorzystywana przy typach sekwencyjnych.

```
>>> len("Python")
6
```

Sfinansowano ze środków Narodowego Funduszu Ochrony Środowiska i Gospodarki Wodnej
Ministerstwo Klimatu i Środowiska
OnGeo

Indeksowanie i segmentacja

Położenie każdego znaku w łańcuchu może być określone za pomocą jego indeksu. Indeks pierwszego znaku ma wartość równą *'0'*, a indeks ostatniego znaku w łańcuchu ma wartość równą *'długość_łańcucha – 1'*. Położenie za pomocą indeksów można podawać także od końca. Wtedy ostatni (pierwszy od końca) znak, przypisany będzie miał indeks *'-1'*, przedostatni *'-2'* itd.

```
>>> s = "PYTHON"
```

s[-8]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
P	Y	T	H	O	N
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]

Ze wszystkich sekwencji w Pythonie, czyli także z łańcuchów znaków, za pomocą indeksów wyodrębniać można segmenty sekwencji. W tym celu wykorzystywana jest następująca składnia:

sekwencja[*start:end:step***]**

Zarówno parametry *start* (początek), *end* (koniec) jak i *step* (krok) są elementami fakultatywnymi. Jeżeli, któryś z nich zostanie pominięty, przyjmowana jest jego domyślna wartość:

start – 0

end – długość sekwencji

step - 1

Poniżej kilka przykładów operacji na segmentach:

```
>>> s[::]  #jeżeli nie podamy żadnych indeksów, wyświetlana jest cała sekwencja
'PYTHON'

>>> s[:4]
'PYTH'

>>> s[3:]
'HON'

>>> s[::2]
'PTO'
```

W charakterze parametrów segmentacji mogą być wykorzystywane także zmienne:

```
>>> a=4
>>> s[a:]
'ON'
```

Łączenie łańcuchów znaków

W języku Python operatorem konkatenacji jest znak **+**. Za jego pomocą można łączyć ciągi znakowe, niezależnie, czy są podane, jako konkretny tekst, zmienna czy segment.

```
>>> s = "Python in QGIS"
>>> s[:-7] + "likes" + s[9:]
'Python likes QGIS'
```

Jeżeli potrzebne jest kilkukrotne powtórzenie tego samego tekstu, można wykorzystać także operator mnożenia *, pamiętając, że łańcuchy zostaną połączone bez żadnych separatorów.

```
>>> s = 'Hop!'
>>> s*4
'Hop!Hop!Hop!Hop!'
```

Innym sposobem łączenia tekstów jest wykorzystanie jednej z wielu metod wbudowanych stringów – metody **join**, której składania wygląda następująco:

'sep'.**join(***string_list***)** – łączy podaną listę ciągów znakowych w jeden

sep – separator jakim mają być rozdzielone elementy listy w wynikowym łańcuchu
string_list – lista ciągów znakowych do połączenia, parametr ten może przyjąć dowolny iterowalny obiekt (np. krotkę)

```
>>> "lody " + '-'.join(['truskawkowo' , 'waniliowo' , 'orzechowe'])
'lody truskawkowo-waniliowo-orzechowe'
```

Dzielenie łańcuchów znaków

.split(*sep [,maxsplit]***)** - rozdziela łańcuch znaków na podstawie danego separatora. Jeżeli separator nie został podany, domyślnie przyjmowana jest spacja.

sep – separator
maxsplit – określa na ile razy maksymalnie, dany ciąg znakowy może być rozdzielony

```
>>> s = "Python in GIS"
>>> s.split()
['Python', 'in', 'GIS']

>>> s.split(' in ')
['Python', 'GIS']
```

.partition(*sep***)** – rozdziela łańcuch tekstowy przy pierwszym wystąpieniu separatora, zwracając trzyelementową krotkę: podłańcuch przed separatorem, separator, podłańcuch po separatorze.

sep – separator

```
>>> s.partition(' ')
('Python', ' ', 'in GIS')
```

```
>>> s.partition('in')
('Python ', 'in', ' GIS')
```

.splitlines(*[keepends]***)** – rozdziela tekst na oddzielne linie na podstawie znaków końca linii.

keepends – przyjmuje wartości True – jeżeli znak końca linii ma być zachowany, False – jeżeli wynik ma być pozbawiony znaków końca linii. Domyślnie wartość parametru przyjmowana jest jako False

```
>>> p = "Pierwsza linia.\n\

Druga linia.\n\
Trzecia linia.\n\
Czwarta linia."


>>> p.splitlines()

['Pierwsza linia.', 'Druga linia.', 'Trzecia linia.', 'Czwarta linia.']


>>> p.splitlines(True)

['Pierwsza linia.\n', 'Druga linia.\n', 'Trzecia linia.\n', 'Czwarta linia.']
```

Wyszukiwanie i zastępowanie podłańcuchów

.find(*sub [,start [,end]]***)** – wskazuje położenie danego znaku lub łańcucha w innym łańcuchu znaków. Jeżeli szukany tekst nie zostanie odnaleziony, zwracana jest wartość '-1'.

sub – szukany tekst
start – początek przeszukiwania łańcucha znaków (indeks początkowego znaku)
end – koniec przeszukiwania łańcucha znaków (indeks końcowego znaku)

```
>>> adr = "12-02-003-0002"

>>> adr.find('-')

2
>>> adr.find('-',3)

5
```

.index(*sub[,start,[end]]***)** – działa analogicznie do **.find**, z tą różnicą, że jeżeli tekst nie zostanie odnaleziony - generowany jest wyjątek.

sub – szukany tekst
start – początek przeszukiwania łańcucha znaków (indeks początkowego znaku)
end – koniec przeszukiwania łańcucha znaków (indeks końcowego znaku)

```
>>> adr.index(';')
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    adr.index(';')
ValueError: substring not found
```

.count(*sub[,start,[end]]***)** – zlicza ile razy dany tekst pojawia się w podanym łańcuchu znaków. Parametry przyjmowane są analogicznie do **.find** .

```
>>> adr = "12-02-003-0002"
>>> adr.count('-')
3
>>> adr.count('-',3)
2
```

.startswith(*prefix[,start,[end]]***)** – zwraca wartość *True* jeżeli łańcuch zaczyna się podanym przedrostkiem, *False* w każdym innym przypadku.

.endswith(*suffix[,start[,end]]***)** – zwraca wartość *True* jeżeli łańcuch kończy się podanym przyrostkiem, *False* w każdym innym przypadku.

```
>>> f = 'python_script.txt'
>>> f.endswith('txt')
True
```

.replace(*old,new[,count]***)** – zwraca kopię łańcucha znaków, w której dany podłańcuch jest zamieniony na inny zdefiniowany. Opcjonalnie można maksymalną liczbę zmian w łańcuchu (np. tylko pierwszy średnik ma zostać zamieniony na spację).

```
>>> f = 'python_script.txt'
>>> f.replace('txt','py')
'python_script.py'
```

Inne metody łańcuchów tekstowych

.upper() – zwraca nowy ciąg znakowy, zawierający same wielkie litery.

.lower() – zwraca nowy ciąg znakowy, zawierające same małe litery.

.capitalize() – zwraca nowy ciąg znakowy, w którym pierwsze litery każdego wyrazu są wielkie a pozostałe małe.

.lstrip(*[chars]***)** – usuwa określone znaki z początku (z lewej strony) łańcucha znaków. Domyślne usuwane są spacje.

W przykładzie poniżej usuwane są wszystkie *0* poprzedzające ostatni człon adresu administracyjnego:

```
>>> adr = '12-02-003-0002'

>>> adr[10:].lstrip('0')
'2'
```

.rstrip(*[chars]***)** – usuwa określone znaki z końca (z prawej strony) łańcucha znaków. Domyślnie usuwane są spacje.

.strip(*[chars]***)** – usuwa określone znaki z obu stron łańcucha znaków. Domyślnie usuwane są spacje.

.ljust(*width[,fillchar]***)** – wyrównuje tekst do lewej strony o odpowiednią liczbę znaków, opcjonalnie wypełniając danym znakiem. Domyślnym wypełnieniem jest spacja.

```
>>> "Python".ljust(20,'-')
'Python--------------'
```

.rjust(*width[,fillchar]***)** - wyrównuje tekst do prawej strony o odpowiednią liczbę znaków, opcjonalnie wypełniając danym znakiem. Domyślnym wypełnieniem jest spacja.

```
>>> "Python".rjust(20,'-')
'--------------Python'
```

.center() – zwraca wyśrodkowany łańcuch znakowy w tekście o zadanej długości.

```
>>> "Python".center(20,'-')
'-------Python-------'
```

Formatowanie tekstów

Formatowanie ciągów tekstowych w języku Python często odbywa się z wykorzystaniem metody format.

.format(**args, **kwargs*) – formatuje dany ciąg tekstowy. Jako parametry podawane są argumenty podstawiane do łańcucha znaków. Nawiasy klamrowe wewnątrz formatowanego ciągu tekstowego traktowane są jako znaki specjalne, umieszczane są w nich indeksy lub nazwy poszczególnych argumentów metody format.

```
>>> tekst1 = 'prosty'
>>> tekst2 = 'przejrzysty'
>>> tekst3 = 'przyjemny'
>>> tekst4 = 'Python jest {0}, {1} i {2}'.format(tekst1,tekst2,tekst3)
>>> tekst4

'Python jest prosty, przejrzysty i przyjemny'
```

Innym sposobem formatowania tekstu jest wykorzystanie wyrażenia z znakiem **%**.

```
tekst4 = 'Python jest %s, %s, i %s' % ('prosty','przejrzysty','przyjemny')
```

gdzie:

%s – ciąg tekstowy

%d – liczba całkowita

%f – liczba zmiennoprzecinkowa

```
>>> tekst4
'Python jest prosty, przejrzysty, i przyjemny'
```

Ćwiczenie 7.

Jak zapisać w stringu:

Pierwszy wers inwokacji Adama Mickiewicza to:

"Litwo! Ojczyzno moja! ty jesteś jak zdrowie"

string_zad_1.py

Ćwiczenie 8.

Utwórz zmienną o nazwie **wiersz** i przypisz do niej następującą sekwencję znaków:

Dęby, buki, świerki i sosny

Przemówcie gromkim głosem

By nikt się nie odważył

Uśmiercić was topora ciosem.

- sprawdź długość wprowadzonego ciągu tekstowego,
- policz ile razy w tekście występuje litera „a”,
- rozbij ciąg tekstowy na poszczególne słowa,
- zamień dęby na brzozy, buki na klony,

string_zad_2.py

Ćwiczenie 9.

Napisz skrypt, który wyświetlał będzie na ekranie wiersz z poprzedniego zadania, ale zamiast słów: *Dęby*, *buki*, *świerki*, *sosny* wypisywane będą słowa podane przez użytkownika.

string_zad_3.py

LISTY (LISTS)

Jednym z podstawowych typów danych, wykorzystywanych przy pracy z Pythonem, są listy – uporządkowane zbiory obiektów. Listy należą do zmiennych i sekwencyjnych typów danych. W jednej liście przechowywać można różne typy danych.

Nową listę definiujemy wymieniając po przecinku poszczególne obiekty listy, zamknięte w nawiasie kwadratowym.

Do poszczególnych elementów listy odwoływać można się z wykorzystaniem indeksowania i segmentowania omówionego w podrozdziale poświęconym pracy z ciągami tekstowymi.

```
>>> l = [] # pusta lista
>>> l = [1, 2, 3] #lista obiektów liczbowych
>>> l = ['tekst1' , 'tekst2', 'tekst3' ] #lista obiektów tekstowych
>>> l = ['tekst1' , 1 , 'tekst2' , 2] #lista z różnymi typami obiektów (liczbowe i tekstowe)
```

Obiektem listy może być także inna lista.

```
>>> l = ['tekst1' , 1 , [1,2,3]]
```

Tak jak w przypadku wszystkich typów sekwencyjnych, listy można:

- dodawać:

```
>>> l1 = [1 , 2 , 3]
>>> l2 = [4, 5 , 6 , 7]

>>> l1 + l2
[1, 2, 3, 4, 5, 6, 7]
```

- powielać

```
>>> l1*4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- sprawdzać zawartość

```
>>> 1 in l1
True
```

- iterować

```
>>> for element in l1:
	print(element + 2)
3
4
5
```

- sprawdzać długość wykorzystując funkcję **len()**

```
>>> l1 = [ 'a' , 'b' , 'c' , 'd' , 'e' ]
>>> len(l1)
5
```

- uzyskiwać dostęp za pomocą indeksów obiektów

```
>>> lista_1 = ['a' , 'b' , 'c' , 'd' , [1 , 2 , 3 ]]

>>> lista_1[0]
'a'

>>> lista_1[2:]
['c', 'd', [1, 2, 3]]

>>> lista1[4][0]  #dostęp do elementów podlisty
1
```

Podstawowe funkcje i metody list

list(*iterable***)** – konwertuje krotkę na listę.

cmp(*list1 , list2***)** – porównuje dwie listy.

```
>>> list1 = [1 , 2 , 3]
>>> list2 = [4 , 5 , 7]
>>> cmp(list1 , list2)
-1
```

max(), **min()** – sprawdza największy i najmniejszy element listy.

.append(*element***)** – dodaje pojedynczy element do listy, wstawiając go na końcu

```
>>> lista = ['element_1', 'element_2', 'element_3']
>>> lista.append('nowy_element')
>>> lista
['element_1', 'element_2', 'element_3', 'nowy_element']
```

.extend(*list***)** – dodaje inną listę do listy. Każdy element nowej listy jest dodawany jako oddzielny

element do starej listy.

```
>>> lista1 = ['element_1', 'element_2', 'element_3']
>>> lista2 = [1,2,3]
>>> lista1.extend(lista2)
>>> lista1
['element_1', 'element_2', 'element_3', 1, 2, 3]
```

.insert(*index, element***)** – dodaje pojedynczy element do listy i wstawia go pod podanym indeksem.

```
>>> lista = ['Asia', 'Krzysiek', 'Basia', ’Ala’]
>>> lista.insert(2,’Bartek’)
>>> lista
['Asia', 'Krzysiek', 'Bartek', 'Basia', 'Ala']
```

.pop() – usuwa element o podanym indeksie oraz zwraca jego wartość.

```
>>> lista = ['Asia', 'Krzysiek', 'Basia', ’Ala’]
>>> lista.pop(0)
'Asia'
>>> lista
['Krzysiek', 'Bartek', 'Basia', 'Ala']
```

.remove(*element*) – wyszukuje podany element w liście i usuwa pierwszy znaleziony.

```
>>> lista = ['Asia', 'Krzysiek', 'Basia', 'Krzysiek', ’Ala’]
>>> lista.remove('Krzysiek')
>>> lista
['Asia', 'Basia', 'Krzysiek', ’Ala’]
```

.count(*value*) – zwraca liczbę wystąpień podanej wartości w liście

```
>>> lista = ['Asia', 'Krzysiek', 'Basia', 'Krzysiek', ’Ala’]
>>> lista.count('Krzysiek')
2
```

.index(*value*) – zwraca indeks pierwszego wystąpienia danej wartości w liście, lub wyjątek jeżeli podana wartość nie znajduje się w liście

```
>>> lista = ['Asia', 'Krzysiek', 'Basia', 'Krzysiek', ’Ala’]
>>> lista.index('Krzysiek')
1
```

.sort() – sortuje elementy wskazanej listy – sortowany jest istniejący obiekt.

```
>>> lista =['e','s','a','g','d', 'e', 'f']
>>> lista.sort()
>>> lista
['a', 'd', 'e', 'e', 'f', 'g', 's']
```

.reverse() – odwraca kolejność elementów istniejącej listy

```
>>> lista =['e','s','a','g','d', 'e', 'f']
>>> lista.sort()
>>> lista
['f', 'e', 'd', 'g', 'a', 's', 'e']
```

Ćwiczenie 10.

Napisz skrypt, który:

- stworzy listę z wartości podanych przez użytkownika,

- posortuje elementy stworzonej listy,

- wyświetli na ekranie ponumerowane kolejno elementy listy.

listy_zad_1.py

Ćwiczenie 11.

Napisz skrypt, który bez użycia funkcji **len()** :

-sprawdzi ile elementów zawiera zdefiniowana wcześniej lista, jeśli lista zawiera inne listy: sprawdzi liczbę elementów listy wraz z liczbą elementów list, które zawiera.

- jeżeli elementami listy są inne listy, wyświetli informację o liczbie takich elementów.

len_list.py

Ćwiczenie 12.

Napisz skrypt, który:

- stworzy listę wygenerowanych 10 losowych liczb całkowitych z zakresu 1-100,
- wyświetli na ekranie wartość minimalną, maksymalną i sumę wygenerowanych wartości.

lista_zad_3.py

KROTKI (TUPLES)

Krotki (*ang. tuples*) są uporządkowanymi zbiorami obiektów dowolnych typów Python. Podobnie jak listy należą do typów sekwencyjnych. W odróżnieniu od list, krotki są niezmienne. Można je dodawać, powielać dokładnie tak samo jak w przypadku list, ale wynikiem jest zawsze nowa krotka. Dozwolone jest także zagnieżdżanie w krotkach innych krotek i dowolnych obiektów Python.

Krotki definiowane są tak jak listy, ale zamiast nawiasów kwadratowych ograniczane są nawiasami okrągłymi.

```
>>> t = (1 , 2 , 3 , 4)
>>> t
(1, 2, 3, 4)

>>> t = (6,)  #krotka jednoelementowa
>>> t
(6,)
```

Jeżeli zdefiniujemy obiekt Python nie używając żadnych nawiasów, tak jak poniżej, wynikiem także będzie krotka:

```
>>> t = 1 , 2 , 3 , 4
>>> t
(1, 2, 3, 4)
```

Listę można przekonwertować na krotkę za pomocą funkcji **tuple()** a krotkę na listę za pomocą funkcji **list()**. W obu przypadkach nie są modyfikowane istniejące już obiekty, ale tworzone nowe.

```
>>> t = ('a' , 'b' , 1 , 2)
>>> l = list(t)
>>> l
['a' ,  'b' ,  1 , 2]
```

Pomimo, że krotki należą do typów niezmiennych i nie można ich modyfikować, można modyfikować obiekty zagnieżdżone w krotce, jeżeli są to obiekty typu zmiennego.

```
>>> t = (1, 2 , 3 , 4)
>>> t = (1, [2, 3] , 4)
>>> t[1] = 'a'

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    t[1] = 'a'
TypeError: 'tuple' object does not support item assignment

>>> t[1][0]='a'
>>> t
(1, ['a', 3], 4)
```

W przeciwieństwie do list, krotki, jak wszystkie obiekty niezmienne, mogą być wykorzystywane w charakterze klucza w słowniku (słowniki omówiono w dalszej części skryptu).

SŁOWNIKI (DICTIONARIES)

Słowniki to nieuporządkowane zbiory par obiektów. W przypadku list (gdzie elementy są uporządkowane), dostęp do poszczególnych elementów listy uzyskiwany jest za pomocą indeksów przypisywanych do obiektów listy, zgodnie z tym jak są posortowane. Pracując ze słownikami, do poszczególnych obiektów słownika odwołujemy się za pomocą indeksu – klucza, gdzie kolejność ułożenia par w słowniku nie ma znaczenia.

```
>>> slownik = {'klucz1': 'wartosc1', 'klucz2': 'wartosc2', 'klucz3': 'wartosc3'}
>>> slownik['klucz1']
'wartosc1'
```

W charakterze klucza mogą być wykorzystywane liczby, teksty lub krotki. Raz ustalony klucz pozostaje niezmienny, w przeciwieństwie do wartości, które mogą być modyfikowane. Wartością może być dowolny obiekt lub zestaw obiektów języka Python. W związku z tym wartości mogą zawierać także zagnieżdżone inne słowniki. Wielkość liter w kluczach ma znaczenie.

Słowniki można tworzyć definiując poszczególne klucze i wartości rozdzielone dwukropkiem, w nawiasach klamrowych, tak jak w przykładzie powyżej. Można także wykorzystać funkcję **dict()**.

```
>>> d = dict(klucz1='wartosc1', klucz2='wartosc2', klucz3='wartosc3')
>>> print(d)
{'klucz1': 'wartosc1', 'klucz3': 'wartosc3', 'klucz2': 'wartosc2'}
```

Kluczem może być obiekt każdego niezmiennego typu.

Słowniki należą do typów niesekwencyjnych (w przeciwieństwie do list i ciągów znakowych). Jeżeli jako argument funkcji **len()** podany zostanie słownik, zwracana jest jego liczba elementów, która jest równa długości listy kluczy słownika. Dla przykładu powyżej funkcja len zwróci wartość 3.

```
>>> len(d)
3
```

Przykładem zastosowania słowników może być książka telefoniczna.

```
>>> ksiazka_tel = {'Ania': 608653203, 'Radek': 894264305, 'Darek': 321567932}
>>> ksiazka_tel
{'Darek': 321567932, 'Radek': 894264305, 'Ania': 608653203}
```

Nowe elementy dodawane są do słownika poprzez podanie klucza i odpowiadającej mu wartości w następujący sposób:

```
>>> ksiazka_tel['Agata']= 867345213
>>> ksiazka_tel
{'Darek': 321567932, 'Radek': 894264305, 'Agata': 867345213,
'Ania': 608653203}
```

Podobnie, aby zaktualizować istniejący element, należy podać jego klucz i nową wartość.

```
>>> ksiazka_tel['Agata'] = 333333333
>>> ksiazka_tel['Agata']
333333333
```

Na powyższym przykładzie zaobserwować można, także sposób odwoływania się do poszczególnych pozycji słownika – tak jak w przypadku typów sekwencyjnych podajemy indeks interesującego nas elementu, dla słowników podajemy wartość danego klucza.

Konkretny element słownika można usunąć za pomocą instrukcji del.

```
>>> del ksiazka_tel['Ania']
>>> ksiazka_tel
{'Darek': 321567932, 'Radek': 894264305, 'Agata': 333333333}
```

Zwróć uwagę, że w każdym z przykładów, elementy tego słownika wyświetlane są w innej kolejności. W przeciwieństwie do list i krotek, słowniki są nieuporządkowanymi zbiorami obiektów, dlatego do poszczególnych elementów nie można odwoływać się za pomocą indeksów wskazujących ich pozycję.

Do podstawowych metod słowników należą:

- **.keys**() – zwraca listę kluczy
- **.values**() – zwraca listę wartości
- **.clear**() – czyści cały słownik (ale zmiennej d nie usuwa)

Ćwiczenie 13.

Napisz program który poprosi użytkownika o wprowadzenie ciągu cyfr z przedziału od 0 do 10,
a następnie wyświetli na ekranie słowną reprezentację wprowadzonego ciągu (np. dla ciągu *123* wyświetlone zostaną słowa *jeden dwa trzy*).

ZamianaLiczb.py

Ćwiczenie 14.

Napisz skrypt, który poprosi użytkownika o podanie nazwy państwa, a następnie korzystając ze zdefiniowanego słownika, wyświetli informacje o stolicy danego kraju. Jeżeli państwo przez użytkownika nie jest uwzględnione w słowniku, niech program wyświetli odpowiednią informację.

slowniki_zad2.py

Ćwiczenie 15.

Napisz skrypt, który na podstawie zdefiniowanego słownika imion i wzrostu przyporządkowanego do osoby, znajdzie imię (imiona) najwyższej osoby i je wyświetli .

slownik_wzrost.py

2.5. Operacje na plikach

Jednym z wbudowanych typów Python jest typ – **file**. Obiekt typu file to połączenie do pliku znajdującego się fizycznie na dysku.

Plik można otworzyć z poziomu skryptu Python za pomocą funkcji **open()**.

open(*name, [mode], [buffering]***)** – otwiera plik.

> *name – nazwa lub ścieżka do pliku – jeżeli podawana jest sama nazwa, zakładane jest, że plik znajduje się w tej samej lokalizacji co skrypt.*

mode – tryb w jakim plik ma być otworzony

- *r – do odczytu*
- *w – do zapisu*
- *a – do aktualizacji – nowa treśc dodawana jest na końcu pliku*
- *r+- do odczytu i zapisu – istniejąca treść nie jest zmieniana*
- *w+- do zapisu i odczytu – istniejąca treść jest kasowana*
- *a+ - do odczytu i zapisu – istniejąca treść nie jest zmieniana, nowa dodawana jest na końcu*
- *b – tryb binarny*

buffering – tryb buforowania

Za pomocą funkcji **open()** można także tworzyć nowe pliki. Jako parametr *name* należy podać ścieżkę do tworzonego pliku, i otworzyć go w trybie do zapisu.

```
>>> open(r'd:\Szkolenie_PYTHON\DANE\TXT\nowy_plik.txt','w')
```

Jak wszystkie typy obiektowe w Pythonie, pliki także mają charakterystyczne dla nich metody.

.close() – zamyka plik.

```
>>>  file = open(r'd:\Szkolenie_PYTHON\DANE\TXT\nowy_plik.txt','w')
>>>  file.close()
```

W przypadku użycia instrukcji with plik jest zamykany automatycznie (nie zawsze jest to jednak pożądane).

```
file_path = r'd:\Szkolenie_PYTHON\DANE\TXT\test_file.txt'
with open(file_path) as file:
    for line in file:
        print(line)
```

file.py

Zawartość pliku można odczytać także w następujący sposób:

```
test_file = open(r'd:\Szkolenie_PYTHON\DANE\TXT\test_file.txt','r')
for line in test_file:
    print(line)
test_file.close()
```

file2.py

W przykładzie powyżej, po wypisaniu każdej linii pliku na ekranie, plik automatycznie zostaje zamknięty.

.read() – odczytuje pełną zawartość pliku i zwraca jako ciąg znakowy.

.readline() – odczytuje zawartość pliku do pierwszego napotkanego znaku końca linii.

.readlines() – odczytuje zawartość pliku ale linia po linii, zwraca listę ciągów znakowych dla pojedynczych linii.

Informacje odczytywane są z pliku zawsze jako tekst, jako tekst są też do pliku zapisywane.

.write(*string*) – zapisuje podany ciąg znakowy do pliku.

string – ciąg znakowy do zapisania

.writelines(*seq*) – zapisuje listę ciągów znakowych do odrębnych linii pliku.

seq – lista ciągów znakowych do zapisania

W przykładzie poniżej odczytywany jest plik zawierający wykaz punktów z współrzędnymi. Dla każdej linii zamieniany jest separator dziesiętny – z kropki na przecinek. Wprowadzone zmiany zapisywane są w utworzonym wcześniej nowym pliku tekstowym.

```
file_path_1 = r'd:\Szkolenie_PYTHON\DANE\TXT\test_file.txt'
file_path_2 = r'd:\Szkolenie_PYTHON\DANE\TXT\test_file_mod.txt'
```

```
file_1 = open(file_path_1,"r")
file_2 = open(file_path_2,"w+")

for line in file_1:
    print(line)
    file_2.writelines(line.replace('.',',') )

file_1.close()
```

Żeby poprawnie odczytać zmodyfikowany plik, kursor trzeba przestawić na jego początek. Służy do tego metoda **.seek()**.

.seek(*offset*) – przestawia kursor na podaną pozycję w pliku.

Za pomocą fragmentu kodu poniżej, kursor przestawiany jest na początek pliku, po czym odczytywana jest zawartość pliku linijka po linijce.

```
file_2.seek(0) #przestawienie kursora na początek pliku

for line in file_2:
    print(line)

file_2.close()
```

replace_TXT.py

Python zawsze zapisuje do pliku jako tekst. Jeżeli zapisuje się liczby i inne nie tekstowe typy przed zapisem powinny być przekonwertowane do tekstu.

Ćwiczenie 16.

Napisz program zliczający w ilu linijkach w pliku tekstowym: *teksty_re.txt* znajduje się adres leśny.

re_adres_lesny_w_pliku.py

2.6. Błędy oraz obsługa błędów

Jeśli w czasie działania programu pojawi się błąd, interpreter wyświetli informacje o błędzie oraz stos wywołań (wszystkie użyte funkcje prowadzące do miejsca w którym wystąpił błąd). Taki błąd jest nazywany wyjątkiem (ang. exception).

Wystąpi na przykład podczas dzielenia przez zero:

```
>>> 10 * (1/0)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Ostatnia linia informacji o wyjątku informuje o rodzaju błędu oraz jego szczegółach. Powyżej znajduje się informacja w jakim pliku oraz numer wiersza w którym wystąpił błąd.

OBSŁUGA BŁĘDÓW

Obsługa błędów jest wykonywana poprzez umieszczenie kodu pomiędzy słowami kluczowymi **try** i **except**. Algorytm wykonywania kodu jest następujący:

- Jeśli program został wykonany bez błędu, to blok kodu except zostanie pominięty
- Jeśli w bloku po słowie kluczowym try wystąpi błąd, pozostała część kodu jest pomijana i:
  - jeśli typ błędu jest zgodny z typem zadeklarowanych za słowem except to wykonywanie kodu przenoszone jest do tego bloku.
  - w przeciwnym razie błąd nie zostanie obsłużony, powodując zatrzymanie programu.

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

Po słowie kluczowym *except* może być zadeklarowanych wiele typów:

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

Jak również kaskada bloków **except**:

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print("I/O error({0}): {1}".format(e.errno, e.strerror))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    #raise
finally:
    print('Clean-up!')
```

Ostatnia deklaracja **except** może również nie zawierać zdefiniowanego typu błędu. Taka konstrukcja oznacza że każdy typ wyjątku/błędu zostanie obsłużony w tym bloku. Należy jednak używać takiej konstrukcji bardzo ostrożnie, ponieważ może ukrywać przed programistą poważny błąd w programie. Blok **finally** służy do sprzątania zasobów i jest zawsze wykonywany niezależnie od tego czy wystąpił wyjątek.

Wyjątki mogą być również zgłaszane (wywoływane) w sposób celowy:

```
>>> raise NameError('HiThere')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

NAJWAŻNIEJSZE RODZAJE BŁĘDÓW WBUDOWANYCH

`Exception`
Klasa bazowa dla wszystkich klas wyjątków. Wszystkie klasy specjalne, zdefiniowane przez użytkownika powinny dziedziczyć z tej klasy.

`ArithmeticError`
Klasa bazowa dla wszystkich klas obsługujących wyjątki podczas operacji arytmetycznych.

`ZeroDivisionError`
Wyjątek występuje podczas operacji dzielenia lub modulo w której mianownik jest równy zero.

`AttributeError`
Występuje podczas wywołania nieistniejącej metody obiektu.

`IOError`
Występuje gdy operacje Wejścia/Wyjścia (I/O) napotkają na przeszkodę np. podczas próby otwarcia nieistniejącego pliku metodą **open()**, podczas zapisu danych na dysku na którym brak jest wolnego miejsca.

`ImportError`
Występuje podczas nieudanej operacji importu modułu.

`IndexError`
Występuje podczas odwołania się do nieistniejącego indeksu np. w obiekcie typu list.

`KeyError`
Występuje podczas odwołania się do nieistniejącego klucza w słowniku.

`ValueError`
Występuje gdy metoda wbudowana lub funkcja otrzymuje argument o nieodpowiedniej wartości (np. z poza dziedziny funkcji), ale odpowiednim typie.

`SyntaxError`
Błąd składni.

`TypeError`
Występuje gdy funkcja stosuje obiekt o niewłaściwym typie.

`NameError`
Występuje podczas użycia nieistniejącej zmiennej.

3. Funkcje

Jeżeli chcemy dany fragment kodu wykorzystywać wielokrotnie w skrypcie, warto rozważyć wykorzystanie funkcji. Funkcje to nazwane fragmenty kodu, które wykonują dane operacje. Raz zdefiniowaną funkcję można wywoływać wielokrotnie w ramach tego samego skryptu.

Nowe funkcje tworzone są za pomocą instrukcji **def**.

```
def nazwa_funkcji(arg1, arg2):
    wyrażenie
    wyrażenie
    ...
    return wynik
```

Poniżej kilka przykładów.

Definicja oraz wywołanie funkcji nie przyjmującej argumentów:

```
def przywitanie():
    print("Hello World!")
przywitanie()

>>>
Hello World!
```

Definicja oraz wywołanie funkcji przyjmującej 1 argument:

```
def przywitanie2(ile):
    for i in range(ile):
        print("Hello World!")

przywitanie2(2)
>>>
Hello World!
Hello World!
```

Definicja oraz wywołanie funkcji zwracającej więcej niż jeden argument:

```
def pole_obwod(a, b):
    pole = a * b
    obwod = 2 * a + 2 * b
    return pole, obwod

pole , obwod = pole_obwod(2, 3)
```

Instrukcja **return** zwraca podany obiekt. Zwracając więcej niż jedną zmienną stosujemy krotkę. Instrukcji return używa się do wyjścia z funkcji, opcjonalnie może zwrócić w tym momencie wartość.

Definicja oraz wywołanie funkcji przyjmującej dowolną liczbę argumentów. Parametr "*args*" zawiera krotkę ze wszystkimi przekazanymi argumentami:

```
def MyName(*args):
    for s in args:
        print(s)

MyName('S', 'm', 'a', 'l', 'l')
```

```
>>>
S
m
a
l
l
```

4. Klasy

Klasy w języku Python posiadają wszystkie standardowe cechy języka obiektowego,

cechy te to min. **tworzenie klas własnych, dziedziczenie klas oraz tworzenie instancji tych klas**. Klasa określa wspólne własności (atrybuty) oraz zachowanie obiektów (metody - służące do komunikowania się z obiektami).

Tworzenie klasy rozpoczyna się słowem kluczowym class wraz z nazwą klasy a następnie (fakultatywnie) w nawiasach okrągłych deklarujemy z jakich klasy będzie klasa dziedziczyła.

```
class pojazd_kolowy:
    def __init__ (self, nazwa, liczba_kol, max_liczba_pasazerow):
        self.nazwa = nazwa
        self.liczba_kol = liczba_kol
        self.maxliczba_pasazerow = max_liczba_pasazerow
        self.__aktualna_liczba_pasazerow = 0

    def dodaj_pasazera(self):
        if self.__aktualna_liczba_pasazerow < self.maxliczba_pasazerow:
            self.__aktualna_liczba_pasazerow += 1

    def podaj_liczbe_pasazerow(self):
        return self.__aktualna_liczba_pasazerow
```

class_pojazd_kolowy_1.py

```
s = pojazd_kolowy("samochod osobowy", 4, 5)   # implementacja klasy, tworzenie instancji klasy
s.dodaj_pasazera()

print("pojazd: " + s.nazwa + ", aktualna liczba pasazerow: " + str(s.podaj_liczbe_pasazerow()))  # dostep do atrybutu klasy

>>>
pojazd: samochod osobowy, aktualna liczba pasazerow: 1
```

UKRYWANIE DANYCH

Python w sposób ograniczony pozwala na tworzenie zmienny prywatnych. Każdy atrybut o nazwie __nazwaAtrybuty (co najmniej 2 podkreślenia i co najwyżej 1 znak podkreślenia po nazwie atrybutu) jest zdefiniowany jako prywatny. Wszystkie pozostałe atrybuty i metody są publiczne.

```
s.__aktualna_liczba_pasazerow # błąd
```

Jednakże możliwy jest dostęp do takich atrybutów przez konstrukcje: _nazwaKlasy__nazwaAtrybutu:

```
s._pojazd_kolowy__aktualna_liczba_pasazerow # OK
```

WŁAŚCIWOŚCI

Pozwalają na automatyczne wywołanie metod dostępu lub przypisania atrybutu instancji klasy.

```
class drzewostan(object):
    def getage(self):
        return 40 # wart. domyslna
    def setage(self, value):
        print('przypisany wiek:', value)
        self._wiek = value

    wiek = property(getage, setage)

x = drzewostan()
print(x.wiek)  # getage

x.wiek = 1 # setage
```

class_drzewostan_1.py

WŁAŚCIWOŚCI TYLKO-DO-OCZYTU (READ-ONLY)

```
class drzewostan(object):
    def getage(self):
        return 40 # wart. domyslna
    def setage(self, value):
        print('przypisany wiek:', value)
        self._wiek = value

    wiek = property(getage)
```

class_drzewostan_2.py

```
x = drzewostan()
print(x.wiek) # getage
x.wiek = 1 # błąd
```

METODY SPECJALNE (PRZECIĄŻENIE OPERATORÓW)

Dzięki tym metodom obiekty utworzone za pomocą klas **mogą być podawane specjalnym operacjom** tj. dodawanie, odejmowanie etc. Struktura nazwy takiej metody to **__nazwaMetody__** (obustronne dwa podkreślenia). Metody te wywoływane są automatycznie. Na przykład metoda **__add__** jest wywoływana jeśli obiekt pojawi się w wyrażeniu + (dodawanie).

```
class Zdanie(object):
    def __init__(self, value):
        self.data = value
    def __add__(self, v):
        return Zdanie(self.data + v)
```

class_zadanie.py

```
a = "xy"
b = a + "z"
print("suma a+z=" + b)h
>>
suma a+z=xyz
```

Wybrane metody:

Metoda	Operator	Użycie
__add__	'+'	X+Y, x+=y
__sub__	'-'	X-Y, x-=y
__repr__, __str__	Konwersja do stringa	Print('X', str(X)
__setattr__	Przypisanie atrybutu	X.var= value
__len__	Długość, liczba obiektów	len(list), len(string)
__iter__	Iteracja zawartości	For loops
__getitem__	Wywołanie zawartości za pomocą indeksu	X[key]

DZIEDZICZENIE KAS (INTERITANCE)

Dziedziczenie jest mechanizmem tworzenia nowych klas po przez modyfikacje już istniejącej klasy. Pierwotna klasa jest klasą bazową, natomiast wtórna nazywana jest klasą pochodną. Klasa pochodna dziedziczy wszystkie atrybuty klasy bazowej. Klasy z których następuje dziedzicznie (klasy mogą dziedziczyć z wielu klas!), wyszczególnione są w nawiasach okrągłych, umieszczonych bezpośrednio po nazwie naszej klasy.

```
class pojazd_kolowy:
    def __init__ (self, nazwa, liczba_kol, max_liczba_pasazerow):
        self.nazwa = nazwa
        self.liczba_kol = liczba_kol
```

```
        self.maxliczba_pasazerow = max_liczba_pasazerow
        self.__aktualna_liczba_pasazerow = 0


class pojazd_mechniczny(pojazd_kolowy):
    def __init__(self, nazwa, liczba_kol, max_liczba_pasazerow,
rodzaj_slinika):
        pojazd_kolowy.__init__(self, nazwa, liczba_kol,
max_liczba_pasazerow)
        self.rodzaj_slinika = rodzaj_slinika
```

class_pojazd_kolowy_2.py