Typy danych

- string – łańcuchy znaków, zapisujemy pomiędzy znakami ‘' lub "" (muszą być zapisane w jednym wierszu) albo potrójnymi apostrofami (‘’’) czy potrójnymi cudzysłowy (”””) - mogą być pisane w kilku linijkach.
  np. a = 'Ala ma kota'

  a = """Ala
  ma
  dużego
  kota"""

  a = 'Ala\nma\ndużego\nkota'

- integer - całkowite
- float - rzeczywiste (zmiennoprzecinkowe, separatorem dziesiętnym jest .)
- complex – zespolone (liczby postaci a+bj)

Komentarze piszemy po znaku #, jeżeli chcemy automatycznie za komentować kilka linii istniejącego kodu stosujemy kombinacje klawiszy crtl+/ (tej samej kombinacji klawiszy stosujemy jak chcemy usunąć komentarz z kilku linii).

Zmienne

Deklaracja

nazwa_zmiennej = wartość

Usuwanie

del a #usuwa zmienna

Drukowanie

print(nazwa_zmiennej) #drukuje zmienną
print(id(nazwa_zmiennej)) #drukuje adres zmiennej

Deklaracja wielokrotna

zm1, zm2, zm3, …, zmn = wart1, wart2, wart3, …, wartn

Zasady tworzenia zmiennych

Możemy używać „podkreślenia” czyli „_” ale nie wolno używać „minusa” czyli „-”.
Nazwa nie może zaczynać się od cyfry. Cyfry mogą się pojawić w dalszej części nazwy
Dla lepszej czytelności używamy małych liter po znaku „_”

Przykłady tworzenia zmiennych.

```
a = '123456' #to jest łańcuch
b = '$zmienna' #to też jest łańcuch
print(a+b)
c,d = 2, 3.14 #wielokrotna deklaracja zmiennych
wynik = c + d
print(wynik)
e = 3 + 2j
print(e)
```

Działania arytmetyczne

Przykłady działań arytmetycznych

```
a = 8
b = 4
c = 3
dzielenie = a / b
print(dzielenie)
dzielenie = a / c
print(dzielenie)
dodawanie = a + b
print(dodawanie)
dzielenie_calkowite = a // c
print(dzielenie_calkowite)
reszta = a % c
print(reszta)
potega = b ** c
print(potega)
potega = pow(4, 3)
print(potega)
```

Operatory przyrostkowe

```
a = 3
#zamiast pisać a=a+1 można zapisać
a += 1
print(a)
```

Formatowanie łańcuchów podczas wyświetlania

```
#Drukujemy liczby
print('wynik działania jest równy a=%(zm)d' % {'zm':12})
a = 5
b = 3
z = 5 - 3
print('Wynik działania %(z1)d-%(z2)d=%(z3)d' %{'z1':a, 'z2':b, 'z3':z})
```

zm, z1, z2, z3 to są nazwy zmiennych, które będą formatowane i pod które można podstawiać odpowiednie liczby

Inny sposób

```
print('wynik działania jest równy a={0:d}'.format(12))

a = 5
b = 3
z = a - b
print('Wynik działania {0:d} - {1:d} = {2:d}'.format(a, b, z))
```

Zadania.

Zad1. Napisz pierwszy skrypt, w którym zadeklarujesz po dwie zmienne każdego typu a następnie wyświetl te zmienne

Zad2. Stwórz skrypt kalkulator, w którym wykorzystać wszystkie podstawowe działania arytmetyczne

Zad3. Napisz skrypt, w którym stworzysz operatory przyrostkowe dla operacji: +, -, *, /, **, %

Zad4. Napisz skrypt, który policzy i wyświetli następujące wyrażenia:

$$e^{10}$$

$$\sqrt[6]{\ln{(5 + sin^2 8}}$$

$$\lfloor 3,55 \rfloor$$

$$\lceil 4,80 \rceil$$

Zad.5 Zapisz swoje imie i nazwisko w oddzielnych zmiennych wszystkie wielkimi literami. Użyj odpowiedniej metody by wyświetlić je pisane tak, że pierwsza litera jest wielka a pozostałe małe. (trzeba użyć metody capitalize)

Zad.6 Napisz skrypt, gdzie w zmiennej string zapiszesz fragment tekstu piosenki z powtarzającymi się słowami np. „la la la”. Następnie użyj odpowiedniej funkcji, która zliczy występowanie słowa „la”. (trzeba użyć metody count)

Zad.7 Do poszczególnych elementów łańcucha możemy się odwoływać przez podanie indeksu. Np. pierwszy znak zapisany w zmiennej imie uzyskamy przez imie[0]. Zapisz dowolną zmienną łańcuchową i wyświetl jej drugą i ostatnią literę, wykorzystując indeksy.

Zad.8 Zmienne łańcuchowe możemy dzielić wykorzystaj zmienną z Zad. 6 i spróbuj ją podzielić na poszczególne wyrazy. (trzeba użyć metody split)

Zad.9 Napisz skrypt, w którym zadeklarujesz zmienne typu: string, float i szestnastkowe. Następnie wyświetl je wykorzystując odpowiednie formatowanie.

Lista

Lista jest podstawowym kontenerem danych języka python. Możemy w niej przechowywać dane różnego typu.
Deklaracji dokonujemy w następujący sposób:
nazwa_listy = [] – w nawiasach kwadratowych umieszczamy elementy, które chcemy umieścić w liście podczas deklaracji, podczas deklaracji listy można jej nie uzupełniać danymi.

Najpopularniejszy zbiór metod typu listy

- append – dodaje element na koniec listy
- insert – dodaje element na wybrane miejsce listy
- pop – jeżeli nie podamy żadnego indesku usuwa ostatni element z listy, jeżeli podamy indeks z listy usunięty zostanie element na wybranej pozycji
- remove – usuwa z listy pierwszą napotkaną wartość, która została podana jako argument
- del – z listy zostanie usunięty element o podanym indeksie
- extend – dodanie sekwencji do listy, sekwencja zostaje dodana na końcu listy
- reverse – odwraca kolejność listy
- sort – sortowanie

```
lista = ['wyraz', 4.23, 5.6, 2, 10, [4,5,6]]

lista.append('slow')
lista.append(5.75)
print(lista)

lista.insert(1, 'pierwszy')
lista.insert(7, 3)
print(lista)

lista.pop()
print(lista)

lista.pop(2)
print(lista)

lista.remove('pierwszy')
print(lista)

del lista[5]
print(lista)

lista.extend((2, 2, 'n'))
print(lista)
```

lista.reverse()
print(lista)

nowa_lista = [7, 5, 8.2, 1, 2.2, 1.1, 10, 6, 3]
nowa_lista.sort()
print(nowa_lista)

Słownik

Jest kontenerem danych przechowującym zbiór danych w postaci klucz:wartość. Za indeksowanie w słowniku odpowiadają klucze.

#Deklaracja słownika
slownik = {1: 22, 2: 22, 3:33, 4.5:10, "cos":"ktos", 4.6:'wartosci'}

Dodanie klucza wraz z wartością do słownika
slownik['klucz'] = 'wartosc'
slownik['6'] = 1.1

Wyświetlenie wartości klucza
print(slownik[4.5])

Usuwanie ze słownika za pomocą klucza
slownik.pop(4.5)

Wyświetlenie wszystkich kluczy słownika
print(slownik.keys())

Wyświetlenie wszystkich wartości słownika
print(slownik.values())

Usunięcie pary klucz:wartość za pomocą del
del slownik[1]

Wprowadzanie danych

Aby wprowadzić dane należy użyć komendy inputa

napis = input("Wpisz dowolny komunikat: ")

print(napis)

print(type(napis))

Dane, które wprowadzamy za pomocą komendy input są typu string, jeżeli chcemy wprowadzić jakąś liczbę musimy dokonać rzutowania typów na typ int lub float, po jej wczytaniu. Przykłady rzutowań do typu int i float.

liczba = input("Wpisz dowolną liczbę: ")

print(liczba)

print(type(liczba))

liczba = int(liczba) # miejsce rzutowania do int

print(type(liczba))

liczba = input("Wpisz dowolną liczbę: ")

print(liczba)

print(type(liczba))

liczba = float(liczba) # miejsce rzutowania do float

print(liczba)

print(type(liczba))

Do wprowadzania danych możemy użyć także komendy readline() i write(s), musimy pamiętać jednak od zaimportowaniu modułu sys. Przykład wczytania danych za pomocą komendy readline().

import sys as system

system.stdout.write("wpisz dowolny komunikat: ")

napis = system.stdin.readline()

system.stdout.write(napis)

Instrukcja Warunkowa

Składnia

```
if warunek_1:
        Instrukcje_1
elif warunek_2:
        Instrukcje_2:
…
elif warunek_n:
        Instrukcje_n
else:
        Inne_instrukcje]
```

Operatory porównania wykorzystywane w instrukcjach warunkowych

== - operator równości, sprawdza czy x jest równy y

!= - sprawdza czy jeden obiekt różni się od drugiego, sprawdza czy x różni się od y

> - większy niż, sprawdza czy x jest większy od y

< - mniejszy niż, sprawdza czy x jest mniejszy od y

>= - większy niż lub równy, sprawdza czy x jest większy lub równy y

<= - mniejszy niż lub równy, sprawdza czy x jest mniejszy lub równy y

Przykład pierwszy: Pobieramy dwie liczby całkowite i sprawdzamy, która jest większa

```
a = input("podaj pierwszą liczbę: ")
b = input("podaj pierwszą liczbę: ")
a = int(a)
b = int(b)
#przy wyświetlaniu zmieniamy liczbe na string
if a > b:
   print("liczba " + str(a) + " jest większa")
elif a < b:
   print("liczba " + str(b) + " jest większa")
else:
   print("wprowadzone liczby są równe")
```

Przykład drugi: Pobiera dwie liczby całkowite i sprawdza czy liczby są równe:

```
a = input("podaj pierwszą liczbę: ")

b = input("podaj pierwszą liczbę: ")

a = int(a)

b = int(b)

if a == b:

    print("wprowadzone liczby są równe")

else:

    print("wprowadzone liczby nie są równe")
```

W instrukcjach warunkowych możemy używać również operatorów logiczny AND(&) lub OR(|)

Przykład trzeci: Pobieramy cztery liczb całkowite i sprawdzamy czy liczba a jest większa od liczb b i liczba c jest większa od liczby d

```
a = input("podaj pierwszą liczbę: ")

b = input("podaj drugą liczbę: ")

c = input("podaj trzecią liczbę: ")

d = input("podaj czwartą liczbę: ")

a = int(a)

b = int(b)

c = int(c)

d = int(d)

if (a > b) & (c > d):

    print("liczba a jest większa od liczby b i liczba c jest większa od liczby d")

else:

    print("liczba a jest mniejsza od liczby b lub liczba c jest mniejsza od liczby d")
```

Wynikiem działania operatorów porównania i operatorów logicznych jest typ bool czyli TRUE lub FALSE

Instrukcja iteracyjna for

for licznik in sekwencja:
    Instrukcje
[else:
    inne_instrukcje]

Sekwencją może być łańcuch, lista lub krotka. Od obliczenia sekwencji zaczyna się działanie instrukcji iteracyjnej. Licznik przyjmuje wartość pierwszego elementu wykonuje instrukcje, następnie przyjmuje wartość kolejnego elementu itd.

Do utworzenia sekwencji możemy użyć funkcji range:

rang(start, stop, step)

Przykład pierwszy: chcemy wyświetlić liczby od 1 do 5

```
for x in range(1, 6, 1):
  print(x)
```

Przykład drugi: tworzymy swoją listę i chcemy jej użyć jako sekwencji do wyświetlenia wartości

```
lista = ['a', 5, 6, 7.5]
for x in lista:
  print(x)
```

Przykład trzeci: wyświetlamy elementy z utworzonej listy, po zakończeniu pętli wyświetlamy komunikat

```
lista = ['a', 5, 6, 7.5]
for x in lista:
  print(x)
else:
  print("Wyświetlanie zakończone")
```

Instrukcja iteracyjna while

while warunek:
    instrukcje
[else:
    inne_instrukcje]

Przykład pierwszy: wyświetlamy liczby od 0 do 10, po zakończeniu pętli wyświetlamy komunikat ile liczb zostało wyświetlonych

```
z = 0
while z != 10:
```

```
    print(z)
    z += 1
else:
    print("Wyświetlony zostało " + str(z) + " liczb")
```

Instrukcja break i continue

Instrukcje umieszczamy w pętlach. Sterują działaniem pętli.
Break – przerywa działanie pętli w której się znajduje (ale nie wszystkich pętli jeśli pętle zagnieżdżamy)
Continue – kończy przebieg aktualnej iteracji pętli i przechodzi do następnego przebiegu.

Przykład: Sprwadzamy czy różnica między podaną liczbą a liczbą z listy równa będzie 0

```
lista = [4, 6, 2, 3, 5, 9, 1]
print("Podaj liczbę a sprawdzę czy różnica między wpisaną liczbą a liczbą z listy równa się 0")
liczba = input("wpisz swoją liczbe: ")
licznik = 0
while licznik < lista.__len__():
    if int(liczba) - lista[licznik] == 0:
        print(liczba + " - " + str(lista[licznik]) + " = 0")
        break
    else:
        licznik += 1
else:
    print("Żadna z liczb, które są w liści nie dała odpowiedniego wyniku")
```

Instrukcje iteracyjne zagnieżdżenia

Pętle, instrukcje warunkowe możemy umieszczać jedna w drugiej. Nazywamy to zagnieżdżaniem.

Przykład Tworzymy dwie listy i robimy sumę poszczególnych elementów

```
lista = [4, 6, 2, 3, 5, 9, 1]
lista2 = [7, 3, 4, 6]
suma = []
```

```
for a in lista:
    for b in lista2:
        wynik = a + b
        suma.append(wynik)
    print(suma)
```

Obsługa błędów

Mamy 3 rodzaje błędów:
1. Błędy składniowe – powstają gdy piszemy program niezgodnie z gramatyką języka np. błędy w definicjach funkcji, niezamknięte nawiasy czy cudzysłów bez pary.
2. Błędy czasu wykonania – powodują przerwanie lub niewłaściwe działanie. Mamy możliwość ich przechwycenia i wymuszenia odpowiedniej reakcji np. użytkownik podaje litery a miał wpisywać liczby.
3. Błędy logiczne – błędy w algorytmie lub programie. Nie wykrywalne przez interpreter ale możliwe przez człowieka po analizie programu

Obsługę błędów realizujemy przez specjalny blok instrukcji.
Składnia:

```
try:
        instrukcje
except nazwa_bledu_1:
        awaryjne_instrukcje_1
…
[except nazwa_bledu_n:
        awaryjne_instrukcje_n]
[else:
        blok_bez_błędu]
```

Przykład

Podajemy dwie liczby do dzielenia, chcemy wyłapać dzielenie przez 0

```
print("Proszę podać pierwszą liczbę")
licz1 = input()
print("Proszę podać drugą liczbę")
licz2 = input()
try:
    wynik = int(licz1) / int(licz2)
    print("Wynik= " + str(wynik))
```

```
except ZeroDivisionError: #nazwa błędu dzielenia przez zero
    print("Tylko Chuck Norris może dzielić przez zero!")
```

Zadania

Zad 1. Napisz skrypt, w którym tworzysz listę ulubionych sportów, odwróć ją a następnie dodaj mniej lubiane sporty na sam koniec.

Zad 2. Stwórz słownik skrótów powszechnie używanych w gazetach lub artykułach internetowych. Jako klucz przyjmij skrót danego słowa, wartość to rozwinięcie tego skrótu.

Zad 3. Stwórz słownik z ulubionymi grami komputerowymi. Pomyśl, co może być kluczem a co wartością w takim słowniku. Policz liczbę elementów w słowniku.

Zad 4. Napisz skrypt, który pobiera od użytkownika zdanie i liczy wystąpienia litery a. Użyj funkcji input

Zad 5. Napisz skrypt gdzie pobierzesz trzy liczby całkowite, gdzie wykonasz obliczenia: $a^b + c$.
Użyj instrukcji readline() i write()).

Zad 6. Wczytaj trzy liczby całkowite a,b,c i sprawdź, która z nich jest największa. W zależności od wyniku wyświetl odpowiedni komunikat. Użyj zagnieżdżeń.

Zad 7. Napisz skrypt, gdzie stworzysz listę składającą się z liczb typu int i float. Następnie za pomocą użycia pętli for podnieś każdą liczbę do kwadratu.

Zad 8. Napisz skrypt, który za pomocą pętli while pobiera 10 liczb, następnie dodaje do listy tylko parzyste liczby.

Zad. 9.
Napisz skrypt, który liczy pierwiastek z liczby podanej przez użytkownika jeśli użytkownik poda wartość ujemną to powinien być wyłapany błąd.

Python Comprehension

Jest to mechanizm służący do generowania kolekcji (lista, słownik, zbiór) na podstawie jednowierszowej definicji. Równoważne definicje zawsze można podać za pomocą pętli. Czasami zaś wystarczy przepisać na język Python definicję matematyczną zbioru.

Możliwa składnia

```
#Zamiast pisać w pętli
lista = []
for element in zakres:
        if pewien_warunek_na(element):
                lista.append(„Cos sie dzieje z:” + element)
#możemy zapisać w jednej linijce
lista = [„Cos sie dzieje z:” + element for element in zakres if pewien_warunek_na(element)]
```

Przykład pierwszy

A = $\{x^2: x \in <0,9>\}$

B = $\{1,3,9,27,\ldots,3^5\}$

C = {x: x∈ A i x jest liczbą nieparzystą}

W pythonie zapiszemy to:

```
#wersja z pentlą
a = []
for x in range(10):
   a.append(x**2)
print(a)
b = []
for y in range(6):
   b.append(3**y)
print(b)
c = []
for z in a:
   if z % 2 == 1:
      c.append(z)
```

```
print(c)
```

#wersja z python comprehension

```
a = [x**2 for x in range(10)]
b = [3**i for i in range(6)]
c = [x for x in a if x % 2 == 1]
print(a)
print(b)
print(c)
```

Przykład drugi

Chcemy uzyskać liczby parzyste z podanego zakresu

#wersja z pętlą

```
liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista = []
for i in liczby:
    if i % 2 == 0:
        lista.append(i)
print("Liczby parzyste uzyskane z wykorzystaniem pętli")
print(lista)
print()
```

#wersja z python comprehension

```
lista2 = [i for i in liczby if i % 2 == 0]
print(lista2)
```

Przykład trzeci zagnieżdżenia

#wersja z zagnieżdżonymi pętlami

```
lista = []
for i in [1, 2, 3]:
    for j in [4, 5, 6]:
        lista.append((i,j))
```

```
print(lista)
```

#wersja z python comprehension

```
lista2 = [(i,j) for i in [1, 2, 3] for j in [4, 5, 6]]
print(lista2)
```

Przykład czwarty związany ze zamianą klucza z wartością w słowniku

#wersja z pętlą

```
skroty = {"PZU": "Państwowy zakład ubezpieczeń",
        "ZUS": "Zaklad ubezpieczeń społecznych",
        "PKO": "Państwowa kasa oszczędności"}
odwrocone = {}
for key,value in skroty.items():
    odwrocone[value] = key
print(odwrocone)
```

#wersja z python comprehension

```
odwrocone2 = {value: key for key, value in skroty.items()}
print(odwrocone2)
```

Funkcje

W pythonie możemy definiować własne funkcje, które będziemy traktować jak podprogramy lub jak funkcje w matlabie.

Składnia

```
def nazwa_funkcji(arg_pozycyjny, arg_domyslny=wartosc, *arg_4, **arg_5):
        instrukcje
        return wartość
```

Definicja instrukcji to instrukcja która tworzy obiekt. Funkcje możemy wywoływać z argumentami lub bez ale zawsze musimy używać nawiasów (nawet jak nie ma argumentów). Funkcja może zwracać jedną lub wiele wartości, które będą zwrócone jako krotka

Przykład pierwszy

Chcemy zdefiniować funkcje, która będzie obliczać pierwiastki równania kwadratowego

```
def row_kwadratowe(a,b,c):
    delta = b**2 - 4 * a * c
    if delta < 0:
        print("brak pierwiastków")
        return -1
    elif delta == 0:
        print("jedne pierwiastek")
        x = (-b) / (2 * a)
        return x
    else:
        print("dwa pierwiaski")
        x1 = (-b - math.sqrt(delta)) / (2 * a)
        x2 = (-b + math.sqrt(delta)) / (2 * a)
        return x1,x2
print(row_kwadratowe(6,1,3))
print(row_kwadratowe(1,2,1))
print(row_kwadratowe(1,4,1))
```

Przykład drugi

Definiujemy funkcję z wartościami domyślnymi

```
import math

def dlugosc_odcinka(x1 = 0, y1 = 0, x2 = 0, y2 = 0):
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

#wywołujemy funkcje dla wartości domyślnych
print(dlugosc_odcinka())
```

```
#wywołujemy funkcje dla własnych podanych wartości
#są to argumenty pozycyjne czyli ważna jest kolejność podania wartości
print(dlugosc_odcinka(1,2,3,4))

#wywołujemy funkcje podając mieszane wartości
#dwie pierwsze interpretowane są jako x1 i y1 jak podano w definicji funkcji
print(dlugosc_odcinka(2, 2, y2=2, x2=1))

#wywołujemy funkcje podakąc wartości nie w kolejności
print(dlugosc_odcinka(y2=5, x1=2, y1=2, x2=6))

#wywołujemy funkcje podając tylko dwa argumenty a reszta domyślne
print(dlugosc_odcinka(x2=5, y2=5))
```

Przykład trzeci

Symbol * oznacza dowolną ilość argumentów przechowywanych w krotce

```
def ciag(* liczby):
    # jeżeli nie ma argumentów to
    if len(liczby) == 0:
        return 0
    else:
        suma = 0
        #sumujemy elementy ciągu
        for i in liczby:
            suma += i
        #zwracamy wartość sumy
        return suma
```

#wywołanie gdy nie ma argumentów

print(ciag())

#podajemy argumenty

print(ciag(1, 2, 3.5, 4, 5, 6, 7, 8))

Przykład czwarty

** dwie gwiazdki oznaczają że możemy użyć dowolną ilość argumentów z kluczem

```
def to_lubie(** rzeczy):
   for cos in rzeczy:
      print("To jest ")
      print(cos)
      print(" co lubie ")
      print(rzeczy[cos])

to_lubie(slodycze="czekolada", rozrywka=['gry', 'filmy'])
```

Moduły i pakiety

Żeby użyć funkcji matematycznych potrzebowaliśmy zaimportować plik math.

Taki plik nazywa się modułem i są tam zapisane po prostu kody w języku Python. Jeśli takich plików będziemy mieć kilka to możemy utworzyć z nich pakiet.

Import modułów systemowych

Jeden import modułu powinien być w jednej linii np.

```
Import sys
```

Można również zapisać import modułu w postaci:

```
from math import *
```

Import modułu zamieszczamy na początku pliku. Ewentualnie za komentarzami. Zaleca się następującą kolejność importów:

- Biblioteki standardowe
- Powiązane biblioteki zewnętrzne
- Lokalne aplikacje/biblioteki

Tworzenie swojego modułu

- Tworząc swój moduł piszemy funkcje i zapisujemy ją do pliku z rozszerzeniem .py
- Następnie dołączamy do nowego skryptu swój moduł używając instrukcji

Przykład

Zawartość pliku litery, który będzie naszym modułem

Plik litery

```
def wyswietl(a):
    print(a)
def dlugosc(a):
    return len(a)
```

Teraz możemy już wykorzystać funkcje z modułu litery (to będzie nowy skrypt)

```
import litery
a = "Ala ma kota"
litery.wyswietl(a)
print(litery.dlugosc(a))
#wyświetla wszystkie zmienne oraz nazwy modułów, które się w nim znajdują
print(dir(litery))
```

Tworzenie swojego pakietu

Pakiet składa się z kilku modułów i najczęściej zapisywany w określonym folderze, gdzie nazwa folderu oznacza nazwę pakietu. Jeżeli chcemy stworzyć pakiet musimy utworzyć katalog dodać tam moduły a następnie dorzucić pliku o nazwie __init__.py, w którym powinien się znaleźć sposób importu plików. Dla stylu **import pakiet.moduł** plik zostaje pusty dla stylu **from pakiet import *** w pliku zapisujemy zmienną __all__ która zawiera wszystkie moduły, które mogą być zaimportowane.

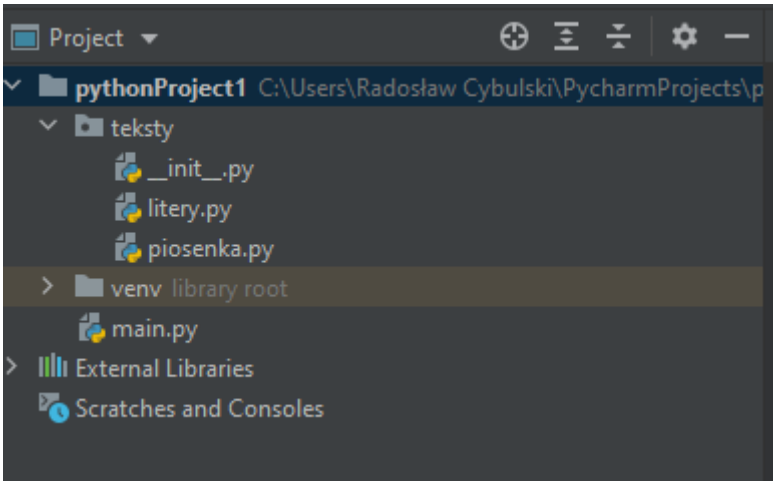
Przykład

Tworzymy jeszcze jeden moduł

#piosenka.py

```
def spiew():
    print("La la la la la")
def zespol():
    print("Boysband")
    print("Girl'n'dance")
```

Tworzymy teraz katalog teksty i wrzucamy tam nasze moduły oraz edytujemy plik __init__.py



Zawartość pliku __init__.py

```
__all__ = ["litery", "piosenka"]
```

Zadania

Zad1

Zdefiniuj następujące zbiory, wykorzystując Python comprehension:

A = {1-x: x∈<1,10>}

B = {1,4,16,…,4^7}

C = {x: x∈B i x jest liczbą podzielną przez 2}

Zad2

Wygeneruj losowo 10 elementów, zapisz je do listy1, następnie wykorzystując Python Comprehension zdefiniuj nową listę, która będzie zawierała tylko parzyste elementy

Zad3

Utwórz słownik z produktami spożywczymi do kupienia. Klucz to niech będzie nazwa produktu a wartość - jednostka w jakiej się je kupuje (np. sztuki, kg itd.). Wykorzystaj Python Comprehension do zdefiniowania nowej listy, gdzie będą produkty, których wartość to sztuki.

Zad4

Zdefiniuj funkcje, która sprawdzi czy trójkąt jest prostokątny.

Zad5

Zdefiniuj funkcje która policzy pole trapezu. Funkcja ma przyjmować wartości domyślne.

Zad6.

Zdefiniuj funkcję która będzie liczyć iloczyn elementów ciągu.

Parametry funkcji a_1 (wartość początkowa), b (wielkość o ile mnożone są kolejne elementy), ile (ile elementów ma mnożyć)

Ponadto funkcja niech przyjmuje wartości domyślne: a = 1, b = 4, ile = 10

Zad7

Napisz funkcje za pomocą operatora *, która wykona te same działanie co w zadaniu 6.

Zad8

Napisz funkcję, która wykorzystuje symbol **. Funkcja ma przyjmować listę zakupów w postaci: klucz to nazwa produktu a wartość to jego koszt. Funkcja ma zliczyć ile jest wszystkich produktów w ogóle i zwracać całościową wartość tych produktów.

Zad9

Stwórz pakiet ciągi. Jeden moduł niech dotyczy działań i wzorów związanych z ciągami arytmetycznymi a drugi niech dotyczy działań i wzorów związanych z ciągami geometrycznymi.

Operacje na plikach

Opercaje na plikach można podzielić na trzy etapy:

1. Otwarcie pliku
2. Działanie na pliku (odczyt lub zapis)
3. Zamknięcie pliku

plik = open(nazwa, tryb, bufor)
gdzie:

plik – nazwa obiektu, którą sami nadajemy
nazwa – nazwa pliku na dysku, jaka jest
tryb – tryb otwarcia pliku (np. do odczytu, do zapisu itd.)
bufor – obszar pamięci przechowujący dane w oczekiwaniu na zapis i odczyt

Wybrane tryby otwarcia plików:
r – tylko do odczytu. Plik musi istnieć:
w – tylko do zapisu. Jeżeli pliku nie ma to zostanie utworzony a jeżeli jest to jego zawartość zostanie zastąpiona nową.
a – do dopisywania. Dane dopisują się na końcu pliku. Jeśli plik nie istnieje to zostanie utworzony
r+ - do odczytu i zapisu. Plik musi istnieć.
w+ - do odczytu i zapisu. Jeśli plik nie istnieje zostanie utworzony.
a+ - do odczytu i zapisu. Jeśli plik nie istnieje zostanie utworzony.

Do odczytania danych z pliku można użyć komend:

- read(rozmiar) – odczytuje dane o rozmiarze, jeśli podany
- readline(rozmiar) – odczytuje wiersze lub ilości znaków jeśli podano rozmiar
- readlines() – odczytuje wiersze z pliku

Przykład 1

```
plik = open("tekst.txt","r")
#odczyt 10 znaków
znaki = plik.read(10)
#odczyt jednej lini z pliku
linia = plik.readline()
#odczyt wierszy z pliku
wiersze = plik.readlines()
#zamknięcie pliku
plik.close()
#drukujemy 10 znaków
print(znaki)
print("\n")
#drukujemy linie
print(linia)
print("\n")
#drukujemy wiersze
print(wiersze)
```

Uwaga 1

Jeśli otwieramy plik i odczytujemy z niego dane jak wyżej to wskaźnik aktualnej pozycji w pliku się przemieszcza. Dlatego w wyniku najpierw otrzymamy pierwsze 10 znaków, potem następne znaki z pozostałej linijki a na koniec resztę linijek tekstu z pliku.

Uwaga 2

Po zakończeniu działania skryptu wszystkie otwarte pliki zamykane są automatycznie.

Do zapisywania danych do pliku możemy użyć:

- write(łańcuch) – zapisuje dane ze zmiennej łańcuch
- writelines(lista) – zapisuje dane z listy

Przykład 2

```
import sys
print("Podaj kierunek studiów, rok i specjalność")
#odczyt danych ze standardowego wyjścia
dane = sys.stdin.readline()
#otwarcie pliku
plik = open("dane.txt","w+")
#zapisanie do pliku
plik.write(dane)
#zamykamy plik
plik.close()
#tworzymy liste
lista = []
for x in range(6):
    lista += [x]
#otwarcie pliku do dopisania
plik = open("dane.txt","a+")
#zapisujemy
plik.writelines(str(lista))
#zamknięcie pliku
plik.close()
```

Przykład 3

Plik możemy otwierać do zapisu i odczytu za pomocą komendy with, wówczas nie musimy martwić się o zamknięcie pliku. Pętla for pozwala na wyświetlenie pliku linijka po linijce

```
with open("tekst.txt", "r") as plik:
    for linia in plik:
        print(linia,end="")
```

Programowanie obiektowe – wprowadzenie

Programowanie obiektowe – podstawowe terminy

Programowanie obiektowe wymaga zaprojektowania struktury opartej na danych i kodzie. Taka struktura nazywana jest klasą a każdy obiekt stworzony na podstawie tej klasy to instancja (albo wystąpieniem) danej klasy. Dane powiązane z obiektem to będą atrybuty (inaczej własności lub właściwości) a funkcje, które wykonują coś na obiekcie to metody.

Enkapsulacja

Inaczej zwana hermetyzacją. Chodzi o to by tak zdefiniować klasę aby jej metody obsługiwały wszystkie właściwości obiektu i żeby żadna funkcja z zewnątrz nie mogła zmienić właściwości obiektu.

Dziedziczenie

Jeśli po utworzeniu klasy okaże się, że potrzebujemy podobnej. np. chcemy stworzyć szafę grającą, która dodatkowo odtwarza mp3 to możemy zastosować dziedziczenie. Wówczas nowa klasa otrzymuje wszystkie właściwości tej, po której dziedziczy. Oryginalną klasę nazywamy klasą bazową (inne nazwy to nadklasa lub superklasa) a nową klasą pochodną (lub podklasą).

Składnia:

```
class NazwaKlasy[(KlasaBazowa1, KlasaBazowa2, KlasaBazowa3)]:
   instrukcje1
   instrukcje2
   instrukcjeN
```

Przykład 4

Definicja pustej klasy i utworzenie obiektu

```
class PierwszaKlasa:
    """Pierwsza klasa python""" #składnia opisowa, opis tego
co jest umieszczone w klasie

obiekt = PierwszaKlasa()
print(obiekt)
```

Przykład 5

Dodajemy atrybut

```
class PierwszaKlasa:
    """Pierwsza klasa python"""
    atrybut = 54321

obiekt = PierwszaKlasa()
```

```
print(obiekt)
print(obiekt.atrybut)
```

Przykład 6

Dodajemy metodę

```
class PierwszaKlasa:
    """Przykład klasy"""
    atrybut = 54321
    def pierwsza_metoda(self):
        return "Teraz działa pierwsza Metoda"
obiekt = PierwszaKlasa()
print(obiekt)
print(obiekt.atrybut)
print(obiekt.pierwsza_metoda())
```

Przykład 7

Dodajemy atrybut do instancji klasy

```
class PierwszaKlasa:
    """Przykład klasy"""
    atrybut = 54321
    def pierwsza_metoda(self):
        return "Teraz działa pierwsza Metoda"
obiekt = PierwszaKlasa()
print(obiekt)
#drukujemy atrybut
print(obiekt.atrybut)
#drukujemy metodę
print(obiekt.pierwsza_metoda())
#dodajemy atrybut do istniejącego obiektu
obiekt.tekst = "la la la"
print(obiekt.tekst)
#ale go nie będzie w nowej instancji klasy
nowy_obiekt = PierwszaKlasa()
print(nowy_obiekt.tekst)
```

Konstruktory

Konstruktor to specjalna funkcja, która tworzy obiekt. Jeśli nie zdefiniujemy swojego konstruktora, to Python wywoła konstruktor domyślny. W Pythonie konstruktor nie tworzy instancji klasy a nadaje wartości początkowe do obiektu.

Przykład 8

```
class Ksztalty:
    # definicja konstruktora
    def __init__(self, x, y):
        #deklarujemy atrybuty
```

```
        #self wskazuje że chodzi o zmienne właśnie
definiowanej klasy
        self.x=x
        self.y=y
        self.opis = "To będzie klasa dla ogólnych kształtów"
    def pole(self):
        return self.x * self.y
    def obwod(self):
        return 2 * self.x + 2 * self.y
    def dodaj_opis(self, text):
        self.opis = text
    def skalowanie(self, czynnik):
        self.x = self.x * czynnik
        self.y = self.y * czynnik
# Tworzymy obiekt
prostokat= Ksztalty(10,30)
# Sprawdzamy teraz jak działają metody które zwracają wartość
print(prostokat.pole())
print(prostokat.obwod())
# sprawdzamy jak działają metody, które nie zwracają wartości
prostokat.dodaj_opis("Prostokąt")
print(prostokat.opis)
prostokat.skalowanie(0.5)
print(prostokat.x)
print(prostokat.y)
```

Uwaga 1

Self reprezentuje instancje klasy. Używając słowa kluczowego „self” możemy uzyskać dostęp do atrybutów i metod w klasie Python. Tworząc metody w klasie zawsze w parametrach używamy słowa kluczowego self.

Uwaga 2

Niektóre funkcje można poprzedzić znakami __ i dla nas będą miały specjalne znaczenie. Konwencja mówi, że wtedy będą to zmienne lub funkcje prywatne czyli takie, które są widoczne tylko dla jednej klasy i nie mogą być modyfikowane przez funkcje i zmienne z innej klasy. W rzeczywistości jest to tylko umowa bo w Pythonie nie ma prywatnych zmiennych czy funkcji, wszystkie są publiczne.

Przykład 9 Do uwaga 2.

```
class Ksztalty:
    # definicja zmiennej poprzedzonej __
    __jestem_prywatna__ = "xyz"

    # definicja konstruktora
    def __init__(self, x, y):
        # deklarujemy atrybuty
        # self wskazuje że chodzi o zmienne właśnie
definiowanej klasy
```

```
        self.x = x
        self.y = y
        self.opis = "To będzie klasa dla ogólnych kształtów"

    def pole(self):
        return self.x*self.y

    def obwod(self):
        return 2*self.x + 2*self.y

    def dodaj_opis(self, text):
        self.opis = text

    def skalowanie(self, czynnik):
        self.x = self.x * czynnik
        self.x = self.y * czynnik

    #Jakaś funkcja
    def zmieniam_tekst(self, tekst):
        tekst += " to jest to ;)”
        return tekst


# Tworzymy obiekt
prostokat = Ksztalty(10,30)

# Sprawdzmy dostęp do zmiennej prywatnej
print(prostokat.__jestem_prywatna__)

# a może uda nam się jeszcze zmienić wartość?
prostokat.__jestem_prywatna__="na na na"
print(prostokat.__jestem_prywatna__)

# spróbujmy czy nowa funkcja coś może zmienić
print(prostokat.zmieniam_tekst(prostokat.__jestem_prywatna__))
```

Zad. 1
Wygeneruj liczby z przedziału <0,30>, następnie pomnóż je przez 2 i zapisz do pliku

Zad. 2
Odczytaj plik z poprzedniego zadania i wyświetl jego zawartość

Zad. 3
Wykorzystując komendę with zapisz kilka linijek tekstu do pliku a następnie wyświetl je na ekranie.

Zad.4

Stwórz klasę NaZakupy, która będzie przechowywać atrybuty:

- nazwa_produktu, ilosc, jednostka_miary, cena_jed
- oraz metody:
- konstruktor – który nadaje wartości
- wyświetl_produkt() – drukuje informacje o produkcie na ekranie
- ile_produktu() – informacje ile danego produktu ma być czyli ilosc + jednostka_miary np. 1 szt., 3 kg itd.
- ile_kosztuje() – oblicza ile kosztuje dana ilość produktu np. 3 kg ziemniaków a cena_jed wynosi 2 zł/kg wówczas funkcja powinna zwrócić wartość 3*2

Zad.5

Utwórz klasę, która definiuje ciągi arytmetyczne. Wartości powinny być przechowywane jako atrybut. Klasa powinna mieć metody:

- wyświetl_dane – drukuje elementy na ekran
- pobierz_elementy– pobiera konkretne wartości ciągu od użytkownika
- pobierz_parametry – pobiera pierwsza wartość i różnicę od użytkownika oraz ilość elementów ciągu do wygenerowania.
- policz_sume – liczy sume elementow
- policz_elementy – liczy elementy jeśli pierwsza wartość i różnica jest podana

Stwórz instancję klasy i sprawdź działanie wszystkich metod.
Zad. 6

Stwórz klasę Robaczek, która będzie sterować ruchami Robaczka. Klasa powinna przechowywać współrzędne x, y, krok (stała wartość kroku dla Robaczka), i powinna mieć następujące metody:

- konstruktor – który nadaje wartość dla x, y i krok
- idz_w_gore(ile_krokow) – metoda która przesuwa robaczka o ile_krokow * krok w odpowiednim kierunku i ustawia nowe wartości współrzędnych x i y
- idz_w_dol(ile_krokow) – metoda która przesuwa robaczka o ile_krokow * krok w odpowiednim kierunku i ustawia nowe wartości współrzędnych x i y

- idz_w_lewo(ile_krokow) – metoda która przesuwa robaczka o ile_krokow * krok w odpowiednim kierunku i ustawia nowe wartości współrzędnych x i y
- idz_w_prawo(ile_krokow) – metoda która przesuwa robaczka o ile_krokow * krok w odpowiednim kierunku i ustawia nowe wartości współrzędnych x i y
- pokaz_gdzie_jestes() – metoda, która wyświetla aktualne współrzędne Robaczka

Stwórz instancję klasy i sprawdź jak działają wszystkie metody

Programowanie obiektowe - dziedziczenie. Iteratory i generatory.

1. Dziedziczenie

Mając klasę bazową możemy utworzyć klasę pochodną, która będzie dziedziczyć po klasie bazowej czyli będzie miała dostęp do atrybutów i metod z klasy bazowej. W klasie pochodnej można dodać nowe metody lub atrybuty.

Przykład

```
class Kształty():
    #definicja konstruktora
    def __init__(self, x, y):
        #deklarujemy atrybuty
        #self wskazuje że chodzi o zmienne własnie
definiowanej klasy
        self.x = x
        self.y = y
        self.opis = "To jest klasa dla ogólnych kształtów"

    def pole(self):
        return self.x * self.y

    def obwod(self):
        return 2 * self.x + 2 * self.y

    def dodaj_opis(self, text):
        self.opis = text

    def skalowanie(self, czynnik):
        self.x = self.x * czynnik
        self.y = self.y * czynnik

#klasa która dziedziczy po klasie Kształty
class Kwadrat(Kształty):
    def __init__(self,x):
        self.x = x
        self.y = x

# i jeszcze klasa, która dziedziczy po klasie Kwadrat
# bedzie definiwoać figurę złożoną z 3 kwadratów w kształcie
litery L
#  _
# | |_
# |_ _|
class KwadratLiteraL(Kwadrat):
    def obwod(self):
        return 8 * self.x

    def pole(self):
        return 3 * self.x * self.y
```

```
#inicjujemy klasę Kwadrat
kwadrat = Kwadrat(5)

#sprawdzenie metod z klasy bazowej
print(kwadrat.obwod())
print(kwadrat.pole())
kwadrat.dodaj_opis("Nasza figura to kwadrat")
print(kwadrat.opis)
kwadrat.skalowanie(0.3)
print(kwadrat.obwod())
print("")

#inicjujemy klasę KwadratLiteraL
litera_l = KwadratLiteraL(5)
print(litera_l.obwod())
print(litera_l.pole())
litera_l.dodaj_opis("Litera L")
print(litera_l.opis)
litera_l.skalowanie(0.5)
print(litera_l.obwod())
```

2. Przesłanianie metod.

Przykład przesłaniania metody został przedstawiony w przykładzie 1, ale warto dodać, że możemy również przesłaniać metody i zmienne dziedziczone po superklasie bazowej object, czyli tej, po której dziedziczy każdy obiekt w Pythonie. Możemy np. przeciążyć metodę __str__(), która zwraca tekstową reprezentację obiektu i domyślnie wyświetla informację o typie obiektu oraz adresie zajmowanym w pamięci komputera.

Przykład

```
class Kwadrat(Kształty):
    def __init__(self,x):
        self.x = x
        self.y = x

kwadrat = Kwadrat(5)
print(kwadrat)
```

```
class Kwadrat(Kształty):
    def __init__(self,x):
        self.x = x
        self.y = x

    def __str__(self):
        return 'Kwadrat o boku {}'.format(self.x)

kwadrat = Kwadrat(5)
print(kwadrat)
```

W pierwszym przypadku zostanie wywołana metoda __str__() klasy object, bo w żadnej wcześniejszej klasie (Kwadrat, Ksztalty) taka metoda nie została znaleziona (funkcja print() wypisuje string więc najpierw mui nastąpić konwersja dowolnego typu na string).

3. Konstruktor klasy bazowej i dziedziczenie wielokrotne.

Poniższy przypadek pokazuje ponownie dziedziczenie jednokrotne po klasie bazowej, gdzie mamy 3 klasy:

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def przedstaw_sie(self):
        return "{} {}".format(self.imie, self.nazwisko)


class Pracownik(Osoba):
    def __init__(self, imie, nazwisko, pensja):
        # wywołanie konstruktora klasy bazowej
        Osoba.__init__(self, imie, nazwisko)
        # lub drugi sposób
        # super().__init__(imie, nazwisko)
        self.pensja = pensja

    def przedstaw_sie(self):
        return "{} {} i zarabiam {}". format(self.imie,
self.nazwisko, self.pensja)


class Menadzer(Pracownik):
    def przedstaw_sie(self):
        return "{} {}, jestem menadżerem i zarabiam
{}".format(self.imie, self.nazwisko, self.pensja)


jozek = Pracownik('Józef', 'Bajka', 2000)
adrian = Menadzer('Adrian', 'Mikulski', 12000)

print(jozek.przedstaw_sie())
print(adrian.przedstaw_sie())
```

Zwróć uwagę na konstruktor klasy Pracownik, który wywołuje konstruktor bazowej klasy Osoba. Natomiast w definicji klasy Manadzer konstruktora nie ma a mimo to jestem w stanie zainicjalizować obiekt tak jak obiekt Pracownik.

Zwróć uwagę na poniższy przykład dziedziczenia wielokrotnego i konstruktor.

```python
class Osoba:

    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def przedstaw_sie(self):
        return "{} {}".format(self.imie, self.nazwisko)


class Pracownik:

    def __init__(self, pensja):
        self.pensja = pensja


class Menadzer(Osoba, Pracownik):

    def __init__(self, imie, nazwisko, pensja):
        Osoba.__init__(self, imie, nazwisko)
        Pracownik.__init__(self, pensja)

    def przedstaw_sie(self):
        return "{} {}, jestem menadżerem i zarabiam
{}".format(self.imie, self.nazwisko, self.pensja)


adrian = Menadzer("Adrian", "Mikulski", 12000)
print(adrian.przedstaw_sie())
```

4. Iteratory i generatory.

Rozpatrując poniższy fragment kodu:

```
for element in range(1,11):
    print(element)
```

Wszystko raczej jest jasne. Ale skąd pętla for wie jak ma się uniwersalnie zachowywać dla różnych obiektów iterowalnych ? Cały mechanizm jest obsługiwany przez iteratory. W niewidoczny dla nas sposób pętle for wywołuje funkcję iter() na obiekcie kolekcji. Funkcja zwraca obiekt iteratora, który ma zdefiniowaną metodę __next__(), odpowiedzialną za zwracanie kolejnych elementów kolekcji. Kiedy nie ma już więcej elementów kolekcji zgłaszany jest wyjątek StopIteration, kończący działanie pętli for. Można wywołać funkcję __next__() iteratora za pomocą wbudowanej funkcji next().

Przykład

```
imie = "Reks"
it = iter(imie)
print(it)
# na wyjściu <str_iterator object at 0x000001CEB9A2F6D0>
print(next(it))# na wyjściu R
print(next(it))# na wyjściu e
print(next(it))# na wyjściu k
print(next(it))# na wyjściu s
print(next(it))# Traceback (most recent call last):
```

Przykład implementacji własnego iteratora.

```
class Wspak:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]


napis = Wspak('Reks')
print(napis.__next__())
for a in napis:
    print(a)
```

Generatory są prostymi narzędziami do tworzenia iteratorów. Generatory piszemy jak standardowe funkcje, ale zamiast instrukcji return używamy yield kiedy chcemy zwrócić wartość. Za każdym razem kiedy funkcja next() jest wywoływana na generatorze wznawia on swoje działanie w momencie, w którym został przerwany . Poniżej przykład generatora, którego działanie jest podobne do iteratora zaprezentowanego w przykładzie.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]


gen = reverse("Feliks")
print(next(gen))
print("Marek")
print(next(gen))
```

Na wyjściu otrzymamy:

s

Marek

k

Podobny efekt możemy również osiągnąć poprzez wyrażenia generujące.

```
litery = (litera for litera in "Zdzisław")
print(litery)
print(next(litery))
```

Na wyjściu:

<generator object <genexpr> at 0x0000014F5FAF1E40>

Z