

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)

Branch: master Python / 3_Prework / Prework_01.md

[Find file](#) [Copy path](#)

daniel Dodano projekt 3 oraz format do prework_1

6167d8b on Mar 24, 2019

[1 contributor](#)

92 lines (57 sloc) 2.57 KB

[Raw](#) [Blame](#) [History](#)   

Python

Prework do 1 listy zadań

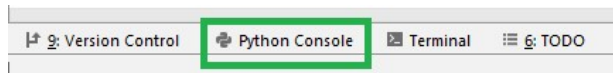
Tryb interaktywny - wprowadzenie

Tryb interaktywny interpretera Pythona jest podstawowym narzędziem nauki i testowania kodu. Jest bardzo podobny do powłoki bash czy powershell. Po prostu wpisujemy komendy w języku Python po czym interpreter je przetwarza i wyrzuca nam wynik podanej komendy. Ot takie proste!

Co ciekawe, po uruchomieniu interpreter wyświetli swoją wersję, opcjonalnie wersję kompilatora C++ (język w którym został napisany Python), informację o sposobie uzyskania pomocy (polecenie help), na końcu zaś znak zachęty >>> lub In[1]. Jeżeli będziemy testować instrukcje złożone, np. warunkowe lub pętle, w interpreterze zobaczymy znaki ... oznaczające, że wprowadzany kod wymaga wcięcia.

Tryb interaktywny - jak uruchomic?

Aby uruchomić tryb interaktywny w programie PyCharm, kliknij Python Console w prawym dolnym rogu.



Powinno się otworzyć okno podobne do poniższego:



Tryb interaktywny - przykładowe użycie

Funkcja print() oznacza wpisanie tekstu na konsolę, let's do this!

```
$ python
Python 2.7.15
>>> print("Siema! To mój pierwszy kod Python!")
Siema! To mój pierwszy kod Python!
>>>
```

To może teraz trochę matematyki? Dodajmy do siebie dwie liczby:

```
>>> 2+2
4
>>>
```

Funkcja print() oraz format()

Funkcja print() służy do wypisywania tekstu na konsolę. Funkcja format() jest bardzo pomocna przy formatowaniu tego co chcemy wyświetlić:

```
>>> print("Hello World!")
Hello World!

>>> kraj = "Polska"
>>> print("Hello {}".format(kraj))
Hello Polska
```

W miejsce klamr {} zostaje wrzucona zawartość zmiennej kraj czyli słowo Polska, a następnie funkcja print() wyświetla sformatowany tekst na konsolę.

Podstawowe typy danych

- typ całkowity (*int*)
- typ zmiennoprzecinkowy (*float*)
- typ logiczny (*bool*)
- typ tekstowy (*str*)
- zmienne są typowane dynamicznie, nie trzeba definiować ich typu, python sam rozpoznaje

```
x = 1 # [zmienna] [operator przypisania] [wartość]
y = 1.0
z = True

# drukuj typy zmiennych x, y i z (oddzielone przecinkiem)
print(type(x), type(y), type(z))
```

```
(<type 'int'>, <type 'float'>, <type 'bool'>)
```

Znak "#" oznacza komentarz czyli fragment kodu w którym może pisać co chcemy a python to zignoruje i przejdzie do kolejnej linii.

To by było na tyle, proste prawda?

W takim razie **zapraszam** na nasze pierwsze zajęcia!

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)

Branch: master Python / 3_Prework / Prework_02.md

[Find file](#) [Copy path](#)

DanielStach Update Prework_02.md

4c418c8 on Mar 17, 2019

[1 contributor](#)

228 lines (141 sloc) 4.44 KB

[Raw](#)[Blame](#)[History](#)

Python

Prework do 2 listy zadań

Łańcuch znaków z ang. (*string*)

Zmienna typu *str* przechowuje tak naprawdę ciąg znaków, liter bądź cyfr, w którym każdy znak ma określoną pozycję. Pozycje natomiast liczymy nie od 1 jak to mamy w zwyczaju ale od 0.

Czyli upraszczając, w programowaniu liczymy od 0 i zapamiętajmy to sobie bo jest to bardzo ważne i bardzo się nam to przyda w przyszłości.

```
s = "Python"
```

```
s[0] # str[i] -> i-ty znak w ciągu
```

```
'p'
```

```
s[1] # pierwszy to drugi? TAK!
```

```
'y'
```

```
s[2]
```

```
't'
```

Litera:	P	y	t	h	o	n
Pozycja/Index:	0	1	2	3	4	5

Lista z ang. (*list*)

- zbiór zmiennych, liczba, danych itd. dowolnego typu
- dynamiczny rozmiar, można ją modyfikować w każdym momencie
- tworzymy ją poprzez:
 - nawiasy kwadratowe
 - rzutowanie *list(jakis_zbior)* (czyli konstruktor klasy *list*)
 - lista składana (*list comprehension*) o tym na kolejnych zajęciach

Lista - przykład

```
lista = [1, 2, 'a', "Python"]
```

```
len(lista) # długość listy
```

```
4
```

```
lista[0] # pierwszy element listy
```

```
1
```

```
lista[-1] # ostatni element listy
```

```
'Python'
```

Krotka z ang. (*tuple*)

- zbiór zmiennych, liczba, danych itd. dowolnego typu
- stały rozmiar - nie można jej zmieniać, tzn. dodawać usuwać elementów
- tworzymy ją poprzez:
 - nawiasy okrągłe
 - ciąg elementów oddzielonych przecinkiem
 - rzutowanie *tuple(jakis_zbior)*

Krotka - przykład

```
krotka = (1, 2, 3, "Python") # jak lista, ale () zamiast []
```

```
print(krotka)
```

```
(1, 2, 3, 'Python')
```

```
krotka = 1, 2, 3, "Python" # brak nawiasów = tuple

print(krotka)

(1, 2, 3, 'Python')

lista = list("Python") # lista ze stringa

krotka = tuple(lista) # krotka z listy

print(lista)
print(krotka)

['P', 'y', 't', 'h', 'o', 'n']
('P', 'y', 't', 'h', 'o', 'n')
```

Lista vs Krotka

- krotka ma stały rozmiar a lista jest dynamiczna
- krotka jest "niezmienna" (*immutable*) w przeciwieństwie do listy (*mutable*)
- więcej o *mutable* vs *immutable* w folderze [4_Teoria](#)
- jeśli sekwencja obiektów jest stała w czasie działania programu, lepiej używać krotek
 - szybsze
 - bezpieczniejsze
 - mogą być kluczami w słowniku (*o słownikach na kolejnych zajęciach*)

Range

- uwaga: w Pythonie 2 `range()` jest funkcją wbudowaną; w Pythonie 3 jest to typ danych (klasa)
- reprezentuje (niezmienniczą) sekwencję liczb
- zajmuje mniej pamięci niż *list* lub *tuple* (przechowuje tylko informację o początku, końcu i kroku)

Range

```
x = range(10) # od 0 do 10

print(x) # w Pythonie 2 zobaczylibyśmy [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Range

```
cyfry = range(0, 10) # range(początek = 0, koniec)
```

```
parzyste = range(2, 10, 2) # range(początek, koniec, krok)
nieparzyste = range(1, 10, 2)

print(cyfry)
print(parzyste)
print(nieparzyste)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8]
[1, 3, 5, 7, 9]
```

Sekwencyjne typy danych

- *list* - dynamiczny ciąg zmiennych dowolnego typu
- *tuple* - niezmienniczy ciąg zmiennych dowolnego typu
- *range* - niezmienniczy ciąg liczb całkowitych
- *str* - niezmienniczy ciąg znaków
- (dla kompletności) są jeszcze binarne sekwencyjne typy danych: *bytes*, *bytearray*, *memoryview*

Indeksowanie

Indeksowanie dla powtórki (bo to bardzo ważne!) zaczyna się od 0 a kończy na $n-1$, gdzie n - długość ciągu.

Długość ciągu w przykładzie poniżej to po prostu ilość liter w słowie "Python" czyli 6, a stosując wzór $n-1$ gdzie n jest równe 6 to $6-1$ równa się 5 :)

Sprawdź czy się zgadza!

```
s = "Python"

n = len(s) # długość łańcucha s

print(n)
```

6

```
s[n-1] # ostatni element
```

```
'n'
```

```
s[n] # poza zakresem
```

IndexError

Traceback (most recent call last)


```
<ipython-input-7-1dea7ae0782c> in <module>()
----> 1 s[n] # poza zakresem
```

IndexError: string index out of range

Litera:	P	y	t	h	o	n
Pozycja/Index:	0	1	2	3	4	5

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)Branch: **master** ▾ [Python](#) / [3_Prework](#) / **Prework_03.md**[Find file](#)[Copy path](#) **daniel** Regulamin

670938a on Oct 17, 2019

[0 contributors](#)

208 lines (139 sloc) 5.85 KB

[Raw](#)[Blame](#)[History](#)

Python

Prework do 3 listy zadań

Pętla - For

Niezwykle prosty mechanizm a bardzo przydatny. Pozwala nam przechodzić czyli "iterować" po każdym elemencie danego zbioru np. listy.

Dla przykładu zdefiniujmy sobie prostą listę i wykorzystajmy pętlę *for* która pozwoli nam wypisać wszystkie elementy tej listy:

```
lista = ['a', 'b', 'c', 'd']

for litera in lista: # czyli w wolnym tłumaczeniu: Dla każdej litery w liście
    print(litera)    # wypisujemy ją na ekran
```

```
a
b
c
d
```

Proste prawda? :)

Pętla - For - enumerate

Teraz dodamy bardzo prostą i fajną funkcję *enumerate* do naszej pętli *for* żeby oprócz literek wyświetlić też pozycje czyli indeks każdej litery. Temat indeksowania powinieneś poznać na poprzedniej lekcji więc to będzie bułka z masłem.

```
lista = ['a', 'b', 'c', 'd']
```



```
for pozycja, litera in enumerate(lista): # czyli w wolnym tłumaczeniu: Dla każdej litery w liście
    print(pozycja, litera)             # wypisujemy ją na ekran
```

```
0, a
1, b
2, c
3, d
```

Pętla - While

Drugi typ pętli, równie prosty jak *for*. Pętli *while* nie powinno się używać na zbiorach jak np. liście, a raczej w przypadkach kiedy chcemy aby wykonywała się jakaś akcja dopóki zdefiniowany warunek jest spełniony.

Nie przejmuj się, to jest bardzo proste, przykład:

```
i = 0

# poniżej po słowie while oznaczającego pętlę mamy warunek który musi być spełniony aby pętla się wykonywała:
while i < 5: # wykonuj dopóki i < 5, w innych słowach wykonuj dopóki i jest mniejsze od 5
    i = i + 1 # gdybyśmy nie dodali tej linijki to mielibyśmy nieskończoną pętlę
    print(i)
```

```
1
2
3
4
5
```

Wzór:

```
while warunek:
    instrukcje
```

Ale proste!

Wyrażenie if

Wyrażenie *if* jest jedną z najłatwiejszych rzeczy w programowaniu a niezwykle przydatną :) otóż założmy że chcemy sprawdzić czy podana liczba jest większa od 0?

Jak to zrobić? Już pokazuję!

```
i = 0
if i > 0:
    print("Zmienna i jest większa od zera!")
else:
    print("Zmienna i jest mniejsza od zera")
```

Kiedy warunek zostanie spełniony zobaczymy napis: "*Zmienna i jest większa od zera!*" natomiast jeśli nasza liczba nie spełni warunku czyli będzie równa 0 bądź mniejsza to zobaczymy napis: "*Zmienna i jest mniejsza od zera*".

Słownik z ang. (*dictionary*) w skrócie *dict*

Ostatni z najbardziej podstawowych zbiorów ale nadzwyczaj użyteczny. Słownik jest zbiorem, którego elementami jest para "klucz": "wartosc_klucza", już tłumaczę na przykładzie o co chodzi.

```
>>> slownik = {"unikanlna_nazwa_klucza": "wartosc_klucza", "unikanlna_nazwa_klucza_2": "wartosc_klucza_2"}
>>> print(slownik)
{'unikanlna_nazwa_klucza_2': 'wartosc_klucza_2', 'unikanlna_nazwa_klucza': 'wartosc_klucza'}
>>> print(slownik["unikanlna_nazwa_klucza"])
wartosc_klucza
>>> print(slownik["unikanlna_nazwa_klucza_2"])
wartosc_klucza_2
```

Warto dodać i zapamiętać, że słownik jest zbiorem mutowalnym - czyli można go modyfikować, a także jest zbiorem nieuporządkowanym - to bardzo ważne!.

Co oznacza nieuporządkowany? Otóż oznacza to że gdy go wyświetlamy funkcją print jego elementy to mogą się one zamienić kolejnością czyli np. pierwsza para może być wyświetlona jako ostatnia:

```
>>> slownik = {"unikanlna_nazwa_klucza": "wartosc_klucza", "unikanlna_nazwa_klucza_2": "wartosc_klucza_2"}
>>> print(slownik)
{'unikanlna_nazwa_klucza_2': 'wartosc_klucza_2', 'unikanlna_nazwa_klucza': 'wartosc_klucza'}
>>> print(slownik)
{'unikanlna_nazwa_klucza': 'wartosc_klucza', 'unikanlna_nazwa_klucza_2': 'wartosc_klucza_2'}
```

Więc skoro słownik jest zbiorem nieuporządkowanym to znaczy, że nie powinniśmy się odwoływać do jego elementów po indeksie ale po nazwie klucza np. slownik["unikanlna_nazwa_klucza_2"].

Jak dodać coś do słownika? Możemy użyć funkcji wbudowanej update() dla dodania kilku par klucz:wartosc na raz:

```
>>> slownik = {"klucz0": 0}
>>> slownik.update({"klucz1": 1, "klucz2": 2})
>>> slownik
{"klucz0": 0, "klucz1": 1, "klucz2": 2}
```

Jak dodać pojedynczą parę?

```
>>> slownik["nowy_klucz"] = "wartość_nowego_klucza"
{"klucz0": 0, "klucz1": 1, "klucz2": 2, "nowy_klucz": "wartość_nowego_klucza"}
```

Pętla - For - dla słownika przy użyciu funkcji items()

Funkcja items() rozbija parę klucz oraz wartość i przypisuje klucz do zmiennej nazwa_klucza a wartość do zmiennej wartosc_klucza, wygląda to następująco:

```
slownik = {"klucz0": 0, "klucz1": 1, "klucz2": 2, "nowy_klucz": "wartość_nowego_klucza"}

for nazwa_klucza, wartosc_klucza in slownik.items():
    print(nazwa_klucza, wartosc_klucza)

klucz0, 0
klucz1, 1
klucz2, 2
nowy_klucz, wartość_nowego_klucza
```

Funkcja do formatowania tekstu - format()

Funkcja format() jest niebiańsko prosta, po prostu wstawia wartość danej zmiennej w miejsce pary klamr "{}", innymi słowy jedna para klamr per jedna zmienna, wygląda to tak:

```
zmienna_int=1
zmienna_string="Kurs"
zmiena_string_2="Python"
print("{} - {} {}".format(zmienna_int, zmienna_string, zmiena_string_2))
```

1 - Kurs Python

Lista składana z ang. (*list comprehension*)

Lista składana to nic innego jak krótsze zapisanie dodawania poszczególnych elementów do zdefiniowanej listy za pomocą pętli for.

Wzór:

[wyrażenie for element in list if warunek]

Przykład:

```
>>> lista = [ i for i in range(10) ]
>>> print(lista)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Pobieranie wartości od użytkownika - funkcja input()

Dzięki funkcji input() jesteśmy w stanie pobrać od użytkownika dowolny ciąg znaków, który może być liczbą, słowem czy nazwą pliku wraz z formatem.

```
>>> x = input("Podaj wartosc x: ")
Podaj wartosc x: 5
>>> print(x)
5
```

To by było na tyle, **zapraszam** na zajęcia!

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)

Branch: master Python / 3_Prework / Prework_04.md

[Find file](#) [Copy path](#)

DanielStach Update Prework_04.md

0f2af16 on Mar 10, 2019

[1 contributor](#)

75 lines (46 sloc) 1.52 KB

[Raw](#)[Blame](#)[History](#)

Python

Prework do 4 listy zadań

Funkcja

Funkcja? Zestaw kodu, który powinien wykonywać konkretną, zdefiniowaną czynność np. dodawać dwie cyfry.

```
def dodaj_dwie_liczby(a,b):  
    print("Dodaje dwie liczby: {} oraz {}".format(a,b))  
    print("Wynik: {}".format(a+b))
```

```
>>> dodaj_dwie_liczby(1,1)  
Dodaje dwie liczby: 1 oraz 1  
Wynik: 2
```

Wzór składni funkcji:

```
def nazwa_funkcji(parametr_1, parametr_n):  
    kod funkcji
```

Funkcje powinny być pisane w sposób *single responsibility* co oznacza, że funkcja powinna wykonywać jedną konkretną czynność. Oznacza to że jeśli funkcja dodaje dwie liczby to wykonuje tylko akcje dodawania, nie powinna mieć opcji odejmowania, dzielenia itd. Umożliwia to utrzymanie tzw. czytelności kodu.

Argumenty Pozycyjne (args)

Argumenty są to wartości przekazywane do funkcji jako zmienne.

Wzór:

```
def funkcja(TUTAJ_DEFINIUJEMY_ARGUMENTY):
```

```
pass
```

```
funkcja(TUTAJ_PRZEKAZUJEMY_WARTOSCI_JAKO_ARGUMENTY_FUNKCJI)
```

Argumenty pozycyjne to te, które są definiowane/przekazywane względem pozycji.

```
def funkcja(param_1, param_2):  
    [pozycja 1], [pozycja 2]
```

Argumenty Kluczowe (kwargs)

Przykład:

```
def funkcja(klucz="WARTOSC_DOMYSLNA_KLUCZA"):  
    print(klucz)  
  
>>> funkcja(klucz="WARTOSC")  
WARTOSC  
>>> funkcja()  
WARTOSC_DOMYSLNA_KLUCZA
```

Argumenty kluczowe to te, które są definiowane/przekazywane względem nazwy tak jak klucz w słowniku.

To by było na tyle, **zapraszam** na zajęcia!

Join GitHub today

[Dismiss](#)

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: **master** ▾ [Python](#) / [3_Prework](#) / **Prework_05.md**[Find file](#)[Copy path](#) **daniel** Prework 6

32c2d60 on Mar 10, 2019

[0 contributors](#)

43 lines (24 sloc) 1.13 KB

[Raw](#)[Blame](#)[History](#)

Python

Prework do 5 listy zadań

Moduł

Pamiętasz jak w liście pierwszej importowaliśmy bibliotekę math? Dzisiaj zrobimy swój własny moduł, który będziemy mogli importować do każdego skryptu kiedy tylko będzie nam potrzebny!

Moduł to jest nic innego jak zwykły skrypt Python'a.

Tworzymy dwa pliki: modul.py oraz skrypt.py

Nasz plik modul.py wygląda następująco:

```
#modul.py

def przykladowa_funkcja():
    print("Jestem funkcja z modulu")
```

A plik skrypt.py wygląda tak:

```
#skrypt.py

import modul

modul.przykladowa_funkcja()
```

Po wywołaniu skryptu skrypt.py otrzymamy:

```
Jestem funkcja z modulu
```

W ten sposób możemy tworzyć pliki python'a które będą zbiorem przeróżnych funkcji, klas, zmiennych i wielu innych składowych, które to w każdej chwili możemy wykorzystać w różnych skryptach na raz!

Takie tworzenie modułów jest jednym z pryncypałów zasady DRY czyli DON'T REPEAT YOURSELF, odnosząca się do tego aby nie pisać tego samego kodu wielokrotnie - nie powtarzać go tylko żeby ten kod był reużywalny.

To by było na tyle, **zapraszam** na zajęcia!

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)Branch: **master** ▾ [Python](#) / [3_Prework](#) / **Prework_06.md**[Find file](#)[Copy path](#)

DanielStach Update Prework_06.md

cbec60d on Mar 10, 2019

[1 contributor](#)

37 lines (21 sloc) 1.37 KB

[Raw](#)[Blame](#)[History](#)

Python

Prework do 6 listy zadań

sys.argv

Możliwe że podczas swojej pracy nad skryptami python'a odnośłeś/odniosłaś w pewnym momencie wrażenie, że dobrzy byłoby sterować działaniem skryptu z poziomu jego uruchomienia.

Załóżmy, że tworzymy skrypt, który czyta dane z pliku, no i super, ale co jeśli nazwa pliku się zmieni? Albo lokalizacja pliku się zmieni?

Pewnie odpowiesz, że "to się łatwo da zmienić w kodzie". A ja powiem, spoko tylko po co? Po co otwierac plik i zmieniać coś w nim? Najgorzej jeszcze jak skrypt dostanie osoba, która nie zna się na programowaniu w pythonie i nie będzie wiedziała gdzie taką nazwę zmienić ale będzie chciała skorzystać ze skryptu.

W takiej sytuacji korzystamy z argumentów skryptu, cos na wzór argumentów funkcji.

Najprostrzym narzędziem do odwołania się do argumentów wywołania skryptu służy lista argv z biblioteki sys.

Przykładowy skrypt:

```
# skrypt.py
import sys
print(sys.argv)
```

Wywołanie skryptu:

```
python skrypt.py 1 2 "trzy" "czwarty_argument" "cokolwiek"
```

Efekt:

```
['skrypt.py', '1', '2', 'trzy', 'czwarty_argument', 'cokolwiek']
```

sys.argv zwraca nam liste której elementami są argumenty przekazane podczas wywołania skryptu oraz nazwa skryptu, która zawsze znajduje się jako zerowy element listy sys.argv.

To by było na tyle, **zapraszam** na zajęcia!

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)Branch: **master** ▾ [Python](#) / [3_Prework](#) / [Prework_07.md](#)[Find file](#) [Copy path](#)

DanielStach Update Prework_07.md

c07326f on Apr 24, 2019

[1 contributor](#)

54 lines (35 sloc) 1.82 KB

[Raw](#)[Blame](#)[History](#)

Python

Prework do 7 listy zadań

Operacje na pliku - czytanie i pisanie

W pewnym momencie na pewno zaczniesz się zastanawiać jak zapisać dane do pliku, a następnie jak je wczytać do skryptu. Służy do tego funkcja *open* oraz wyrażenie *with*.

Wzór:

```
with open(nazwa_pliku, tryb) as dowolna_nazwa_obiektu_pliku:
    dowolna_nazwa_obiektu_pliku.funkcja_obiektu_pliku
```

Przykłady:

```
with open("tekst.txt", "w") as f:
    f.write("przykładowy tekst do zapisu")
```

```
with open("tekst.txt", "r") as f:
    dane = f.read()
```

```
>>> print(dane)
przykładowy tekst do zapisu
```

Podstawowe tryby: r - czytanie (read), w - pisanie (write), a - dodawanie do pliku (add/append)

argparse - czyli lepszy sys.argv

Biblioteka argparse jest rozwinięciem funkcjonalności argumentów dla skryptu. Dzięki tej bibliotece możemy nazywać argumenty i definiować dla nich pomoc tzw. help, definiować czy są wymagane czy opcjonalne, jakie mogą przyjmować wartości i wiele wiele innych!

Przykład:

```
import argparse

def arg_parser():
    parser = argparse.ArgumentParser() # tutaj tworzymy instancje obiektu ArgumentParser
    # ponizej tworzymy argument i odpowiednio nadajemy mu nazwe --mode, potem opisujemy jego krotka pomoc (!
    # definiujemy jego możliwe wartosci za pomocą argumentu choices dla funkcji add_argument
    # Wartosc domyslana rowna 0, typ argumentu rowny int oraz ze parametr nie jest wymagany -> Required=False
    parser.add_argument('--mode', help="Mode how to read file, 0 - (default choice) read whole file, 1 - om:
    return parser.parse_args() # tutaj parsujemy argumenty i zwracamy z funkcji obiekt parser
```

To by było na tyle, **zapraszam** na zajęcia!

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)

Branch: master Python / 3_Prework / Prework_08.md

[Find file](#)[Copy path](#) daniel Prework 8

0568de5 on Mar 10, 2019

[0 contributors](#)

15 lines (9 sloc) 574 Bytes

[Raw](#)[Blame](#)[History](#)

Python

Prework do 8 listy zadań

Samodzielny research

W tym preworku, będziesz miał/miała bardzo proste dwa zadania:

1. Wyszukać w dokumentacji Python'a informacji na temat funkcji `getsizeof` z biblioteki `sys`.
2. Wyszukać w dokumentacji Python'a informacji na temat generatora, czym jest co robi i jak oraz kiedy go stosować.


Przeczytaj to co znalazłeś i wyciągnij wnioski, które przedyskutujemy na zajęciach.

UWAGA: Zapisz linki do dokumentacji jako potwierdzenie, że skorzystałeś z dokumentacji Python'a, a nie sławnego `stackoverflow` ;)

All your code in one place

[Dismiss](#)

GitHub makes it easy to scale back on context switching. Read rendered documentation, see the history of any file, and collaborate with contributors on projects across GitHub.

[Sign up for free](#)[See pricing for teams and enterprises](#)Branch: master ▾ [Python](#) / [3_Prework](#) / Prework_09.md[Find file](#)[Copy path](#) daniel Prework 9 i 10

89813ed on Mar 10, 2019

[0 contributors](#)

11 lines (6 sloc) 411 Bytes

[Raw](#)[Blame](#)[History](#)

Python

Prework do 9 listy zadań

Klasy - zrozumienie przez szukanie

W tym preworku poproszę Cię abyś przeszukał wszelkie dostępne Ci źródła i poszukał wyjaśnienia czym są klasy w programowaniu, nie tylko w Pythonie.

Znajdź przykłady, postaraj się je przeanalizować, to będzie początek i punkt zaczepienia w dyskusji na temat klas na naszych kolejnych zajęciach.

Powodzenia!

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)[Dismiss](#)Branch: **master** ▾ [Python](#) / [3_Prework](#) / **Prework_10.md**[Find file](#)[Copy path](#) **daniel** Prework 9 i 10

89813ed on Mar 10, 2019

[0 contributors](#)

28 lines (19 sloc) 509 Bytes

[Raw](#)[Blame](#)[History](#)

Python

Prework do 10 listy zadań

Global - zmienne globalne

Zmienne globalne to takie, które są dostępne z każdego poziomu skryptu, nawet wewnątrz funkcji.

Przykład:

```
zmienna_globalna=0

def funkcja():
    global zmienna_globalna
    print(zmienna_globalna)

def funkcja1():
    zmienna_globalna = 2 # ale jednak lokalna
    print(zmienna_globalna)

print(zmienna_globalna)
funkcja()
funkcja1()
print(zmienna_globalna)
```

To by było na tyle, **zapraszam** na zajęcia!