
infoShare
ACADEMY
warsztaty
Python
dla początkujących

Hello!!

Maciej Kwiatkowski

maciej.kwiatkowski92@gmail.com

https://github.com/sucharinio

Agenda

- 18:00 - 19:50 Podstawy Python
- 19:50 - 20:00 Przerwa
- 20:05 - 21:00 Przetwarzanie plików tekstowych

Zasoby

- **Google**
- Dokumentacja Python: *https://docs.python.org/3/*
- StackOverflow: *https://stackoverflow.com/*
- GitHub: https://github.com/sucharinio/python_podstawy

1. Podstawy Python

Dlaczego Python?

- Prosta składnia, zbliżona do języka naturalnego
- Kod niezależny od systemu operacyjnego (bo interpretowany)
- Wszechstronność (Big Data, AI, Web development, pentesting)
- Popularność

Tworzenie kodu

- Interpreter
- Edytor kodu:
  - zwykły edytor tekstu (pliki z rozszerzeniem *.py)
  - IDE (*Integrated Development Environment*):
    - podpowiedzi,
    - kolorowanie składni,
    - debugger,
    - automatyzacja uruchamiania (kodu, testów),
    - wtyczki (repozytorium kodu, klient bazy danych)

Interaktywna konsola

- Wykonuje kod linijka po linijce.
- Bardzo przydatna do testowania rozwiązań “na boku” lub sprawdzenia działania/wywołania funkcji.

Podstawowe typy danych

- **`None`** – nic, null, brak
- 123 - **`int`** - liczby całkowite (integer)
- 54.45 - **`float`** - liczby zmiennoprzecinkowe (floating point)
- "Ala #23" - **`str`** - łańcuchy znaków (string)
- True/False - **`bool`** - prawda/fałsz (boolean)

Zmienna

- Nazwany obszar pamięci, w którym znajduje się jakaś wartość.
- Pozwala na ponowne użycie wartości w innym miejscu w kodzie.
- Zbiór dostępnych zmiennych nazywamy **przestrzenią nazw**.
- Przypisanie realizowane jest przez pojedynczy znak równości “=”.

```
>>> my_value = 124
>>> nazwisko = "Kowalski"
>>> czy_obecny = True
```

Operatory

Operatory

Matematyczne:

```
+, -, *, /, //, %, **
```

Logiczne:

```
==, !=, <, >, <=, >=, in, is, and, or, not
```

Przypisanie, kolejność wykonywania działań

=

- Python wykonuje wyrażenia od lewej do prawej.
- W przypadku przypisywania,
  najpierw wykonywane są operacje z prawej strony znaku przypisania (obliczenie wartości),
  później z lewej strony.
- Kolejność wykonywania można sprawdzić tutaj:
  https://docs.python.org/3/reference/expressions.html#operator-precedence

```
>>> wynik = 5 != 4 and 'a' not in 'Andrzej'
```

Operator porównania

= VS ==

- **=** przypisuje wartość do zmiennej

```
>>> x = 1
```

- **==** porównuje dwie wartości (zwraca True lub False)

```
>>> 1 == (2 - 1)  # True
```

Komentarze

#

Wszystko po tym znaku jest ignorowane przez interpreter.
Może służyć do opisania fragmentu kodu lub jego “*wykomentowania*” (tymczasowego usunięcia).

Przykłady

Pliki:

- czesc_1/00_typy.py
- czesc_1/01_zmienne.py

Funkcja

- Blok wydzielonego kodu, używany do wykonania określonego działania.
- Funkcje zapewniają modułowość kodu oraz ułatwiają jego ponowne użycie.
- Python posiada funkcje wbudowane oraz pozwala na ich definiowanie.
- Wywołanie funkcji:

```
nazwa_funkcji()
nazwa_funkcji(argumenty)
>>> type(“Ala ma kota”)
```

Metody wbudowane typów

Każdy typ danych (*string*, *integer*) posiada zdefiniowane metody (funkcje), które pozwalają na wykonanie różnych (najpopularniejszych) działań, właściwych dla tego typu.

```
typ.funkcja()
>>> ”ala ma kota”.capitalize()
```

Funkcje input i print

```
>>> nazwisko = input(”Podaj nazwisko: ”)
```

input(prompt_text) przyjmuje od użytkownika dane i zapisuje je do zmiennej o typie string.

```
>>> print(nazwisko)
```

print() służy do wypisania tekstu na ekranie.
Automatycznie dodaje na końcu stringa znak specjalny nowej linii *\n*.

Rzutowanie, czyli konwersja typu

- Każda zmienna posiada **typ**, który określa rodzaj informacji w niej przechowywanej oraz operacje jakie można na niej wykonać.
- Czasami potrzebna jest zmiana typu zmiennej (rzutowanie, *cast)* w celu wykonania na niej operacji charakterystycznej dla typu np. zamiana *wejścia* użytkownika ze *stringu* na *inta* w celu wykonania operacji arytmetycznej.

```
nazwa_typu(wartosc)
>>> int(“4”)  # 4
>>> str(4)  # “4”
```

Rzutowanie c.d.

- Rzutowanie może powodować utratę informacji (rzutowanie *floata* na *inta*) lub nawet błąd (rzutowanie litery na liczbę całkowitą).

```
>>> int(2.5) # 2
>>> int(“Ala”)  # ValueError
```

Przykłady

Pliki:

- czesc_1/02_int_string.py
- czesc_1/03_input.py

Lista

```
list(), []
```

- Zbiór kolejnych par indeks - wartość, mogących różnić się typem wartości.
- W Pythonie indeksy są liczbami całkowitymi, pierwszym indeksem jest zawsze 0 (zero).
- Do elementu odwołujemy się przez indeks.
- Możliwe jest wycinanie fragmentu listy (*slice*), który staje się **nową** listą.

Przykłady list

```
>>> lista = [1, 2, 3]
>>> lista2 = ["owca”, “lama”, “stado”]
>>> lista3 = []  # pusta lista
>>> lista4 = [1, "dwa", 3, 4]
>>> lista5 = list(range(3))  # [0, 1, 2]
```

Range

Funkcja zwracająca kolejne wartości z zadanego przedziału.

```
range(stop)
>>> range(3)  # <0, 1, 2>
range(start, stop)
>>> range(4, 8)  # <4, 5, 6, 7>
range(start, stop, krok)
>>> range(0, 10, 3)  # <0, 3, 6, 9>
```

Operacje na listach

- Sprawdzanie długości:

```
>>> len([0, 1, 2])  # 3
>>> lista2 = ["owca”, “lama”, “stado”]
```

- Wyciągnięcie elementu o indeksie 1:

```
>>> lista2[1]  # “lama”
```

- Przypisanie do indeksu 0 nowej wartości

```
>>> lista2[0] = “wataha”
```

- Dodanie nowego indeksu do listy:

```
>>> lista2.add(“grupa”); print(lista2)
["owca”, “lama”, “stado”, “grupa”]
```

- **String** wspiera niektóre operacje list (indeksowanie, długość)!

Krotka

```
tuple(), (,)
```

Niezmienna (zarówno pod względem długości jak i zawartości) wersja listy.

```
>>> tuple1 = ("raz", "dwa", "trzy")
>>> tuple1[0] = "jeden"  # TypeError
>>> x = "raz",  # tuple
>>> x1 = (“raz”,)  # tuple
>>> x2 = ("raz")  # string
```

Słownik

```
dict(), {klucz1: wartosc1, ...}
```

- Zbiór par klucz - wartość, mogących różnić się typem wartości.
- Klucz musi być typem niezmiennym (np. string, int, tuple) i być unikatowy (tylko jeden wewnątrz słownika).
- Klucze nie zachowują kolejności (alfabetycznej, podania).

Przykłady

Pliki:

- czesc_1/04_list_range.py
- czesc_1/05_tuple.py
- czesc_1/06_dict.py

Blok kodu

Dwukropek rozpoczynający blok

```
Instrukcja/wyrazenie:
    Instrukcja
    Instrukcja
    Instrukcja/wyrazenie:
        Instrukcja
    Instrukcja
    I tak dalej...
```

Poziom 1
(4 spacje)

Poziom 2.
(8 spacji)

Blok kodu

- W Pythonie fragmenty kodu wydzielane są przez wcięcie tekstu (indentacje), nie klamry/znaki specjalne.
- Instrukcje poprzedzone takim samym wcięciem z białych znaków (tabulacji lub spacji) zawierają się w jednym bloku (dotyczą jednego zbioru operacji np. w pętli).
- Ważne jest aby wcięcie było zawsze konsekwentnie stosowane (rodzaj/liczba znaków).
- PEP 8 (Python Enhancement Proposal, *https://www.python.org/dev/peps/pep-0008/*) sugeruje używanie 4 spacji jako pojedynczego wcięcia.

Instrukcja warunkowa

```
if warunek:
    # kod wykonany gdy warunek prawdziwy
elif inny warunek:
    # kod wykonany gdy warunek w if był falszywy
    # warunek w tym elif musi byc prawdziwy aby ten kod wykonac
elif (inny warunek1 and inny warunek2):
    # elif-ów moze być wiele lub nie byc zadnego,
    # kod wewnatrz elif wykona sie tylko gdy wszystkie
    # poprzedzajace go warunki były niespelnione (falszywe)
else:
    # przypadek domyslny, nie ma warunku,
    # kod w else bedzie wykonany gdy wszystkie poprzednie warunki
    # byly fałszywe, else moze byc tylko jeden lub wcale
```

Pętla while

```
while (warunek):
    blok kodu
```

- Kod w pętli `while` wykonywany będzie tak długo jak długo warunek będzie spełniony (rzutowalny na `True`).
- Oznacza to że warunek może być niespełniony już na początku (pętla nie wykona się nigdy).
- Kod wewnątrz pętli while, będzie powtarzany dopóki wartość logiczna (wyrażenia lub zmiennej) nie zmieni się na `False`*.

** chyba, że pętla zostanie przerwana lub zmodyfikowana*

Pętla for

```
for element in kolekcja:
    blok kodu
```

- Pętla `for` wykona się dla każdego elementu w kolekcji*.
- Przestrzeń nazw bloku kodu jest rozszerzana o zmienną `element` (nazwa zmiennej może być dowolna).
- Kolekcja może być pusta.
- Pętle można zagnieżdżać (dotyczy również pętli `while`).

** chyba, że pętla zostanie przerwana lub zmodyfikowana*

Zmienianie przebiegu pętli

`continue` – program pomija pozostałe instrukcje w bloku i wraca do sprawdzenia warunku (`while`) lub do kolejnego elementu (`for`).

`break` – działanie pętli jest przerywane, program przechodzi do kolejnej instrukcji po całym bloku pętli.

Przykłady

Pliki:

- czesc_1/07_while.py
- czesc_1/08_for.py

Funkcja

- Blok wydzielonego, powtarzalnego kodu, używany do wykonania określonego, *możliwie prostej* działania.
- Funkcje zapewniają modułowość kodu oraz ułatwiają jego ponowne użycie (*zasada DRY*).
- Python posiada funkcje wbudowane oraz pozwala na ich definiowanie.
- Wywołanie funkcji:

```
nazwa_funkcji()
nazwa_funkcji(argumenty)
type(“Ala ma kota”)
```

Definicja funkcji

```
def nazwa_funkcji(parametr1, ...):
    Instrukcja
    Instrukcja
    Instrukcja/wyrazenie:
        Instrukcja
    Instrukcja
    return wartosc
```

Nagłówek funkcji
(nazwa + parametry)

Ciało
funkcji

Wartość zwracana

Definicja funkcji - słowa kluczowe

- **Definicja funkcji** musi zaczynać się od wyrażenia *def*.
- **Nazwa funkcji** pojawia się w przestrzeni nazw.
- Funkcja posiada od zera, do nieskończenie* wielu **parametrów**.
- **Ciało funkcji** tworzy swoją przestrzeń nazw, przez rozszerzenie przestrzeni nazw, w której jest zdefiniowana o parametry z **nagłówka funkcji** (mogą nadpisać wcześniejsze zmienne).
- **Funkcja “zawsze coś zwraca”**, choć nie zawsze musi posiadać wyrażenie *return*. Może zawierać ich wiele, lecz instrukcje w bloku po wyrażeniu *return* nigdy nie będą wykonane.

Argumenty funkcji

- Funkcja może nie posiadać żadnych argumentów.

```
>>> def funkcja_1():
        return 5
>>> funkcja_1()  # 5
```

- Paramtery funkcji przypisywane są kolejno przy wywołaniu.

```
>>> def funkcja_2(x, y):
        return x + y
>>> funkcja_2(4, 2)  # 6, x = 4, y = 2
```

Argumenty funkcji

- Jeżeli podamy mniej wartości niż funkcja ma parametrów, interpreter zgłosi błąd.

```
>>> def funkcja_2(x, y):
        pass
>>> funkcja_2(4)  # TypeError
```

- Jeżeli podajemy parametry po nazwach, możemy zmienić ich kolejność.

```
>>> def funkcja_2(x, y):
        pass
>>> funkcja_2(y=4, x=2)  # x = 4, y = 2
```

Argumenty funkcji

- Jeżeli funkcja ma wiele parametrów, możemy wywołać ją w sposób mieszany, część parametrów podając jawnie (*explicit*), *część nie*.

```
>>> def funkcja_3(x, y, z):
        pass
>>> funkcja_3(4, z=1, y=3)
```

- Należy wtedy pamiętać, że nie możemy podać parametru więcej niż raz, oraz podawać paramtery pozycyjnie (niejawnie) za jawnymi.

```
>>> funkcja_3(4, z=1, x=3)  # TypeError
>>> funkcja_3(y=4, x=2, 5)  # SyntaxError
```

Argumenty domyślne

Możliwe jest przypisanie domyślnej wartości parametru funkcji. Wartość ta będzie przypisana, gdy funkcja zostanie wywołana z mniejszą liczbą parametrów (jednak wszystkie parametry pozycyjne muszą być zapewnione). **Nie powinno się używać kolekcji mutowalnych** (np. list) **jako parametrów domyślnych**.

```
>>> def funkcja_4(x, y, z=1):
        pass
>>> funkcja_4(4, 5)  # x=4, y=5, z=1
>>> funkcja_4(4, 5, 6)  # x=4, y=5, z=6
>>> funkcja_4(4)  # TypeError
```

Wartość zwracana

- Jeżeli chcemy, żeby funkcja zwróciła jakąś wartość używamy wyrażenia *return wartosc*.
- Możemy użyć również samego wyrażenia *return*, wówczas zwracana jest wartość **None**.
- Funkcja nie posiadająca w swoim ciele wyrażenia *return* również zwraca wartość **None**.
- Wartość/wartości zwracane przez funkcję możemy przypisać do zmiennej.

```
>>> def square(x):
        return x * x
>>> z = square(4)  # z = 16
```

Przykłady

Pliki:

- czesc_1/09_funkcje.py
- czesc_1/10_funkcje.py
- czesc_1/11_funkcje.py

Obiekt z klasą ;)

- Obiekt to abstrakcyjne pojęcie zbioru danych (***atrybutów***) i funkcji (***metod***), które mogą te dane modyfikować lub używać.
- Definicja typu opisana jest za pomocą ***klasy***.
- Pojedynczy obiekt z określoną wartością atrybutów jest ***instancją klasy***.
- W Pythonie ***wszystko*** jest obiektem (nawet klasa).

Definicja klasy

```
class NazwaKlasy(rodzic1, ...):
    wartosc = 1
    def __init__(self, name):
        self.name = name
        print("I'm alive!")

    def is_alive(self):
        print(
            f"{self.name} "
            "is still alive."
        )
```

Nagłówek klasy
(nazwa + rodzice)

Atrybut klasy

Metoda magiczna
inicjalzująca
instancję

Atrybut
instancji

Inna metoda klasy

Metody klasy

```
def rename(self, name):
    self.name = name
    print(f“My name is now: {self.name}”)
```

- Pierwszy parametr każdej metody klasy będzie wypełniony referencją do instancji klasy, dlatego wywołując metodę klasy podajemy jeden parametr mniej.
- Atrybuty klas powinny być modyfikowane przez metody klasy.

Przykłady

Pliki:

- czesc_1/12_obiekty.py

Przerwa (15 min)

2. Przetwarzanie plików

Czytanie plików

- Pliki otwierane są za pomocą metody `open`.
- Jedynym parametrem pozycyjnym funkcji jest *`file`*, który określa ścieżkę do pliku. Może być ona względna (względem miejsca uruchomienia programu):
  `/home/<user>/python_podstawy/czesc_2/plik.txt`
  lub bezwzględna (jednoznaczna w przestrzeni komputera):
  `czesc_2/plik.txt`

Czytanie plików

- Metoda zwraca obiekt pliku, domyślnie w trybie czytania.
- Konieczne jest zamknięcie pliku po zakończeniu pracy.

```
_file = open(“plik.txt”)
…
_file.close()
```

- Możemy także skorzystać z wyrażenia `with`, które zamknie plik po wyjściu programu z bloku kodu

```
with open(“plik.txt”) as _file
    …
print(_file.closed)  # True
```

Tryby otwarcia pliku

- Z pliku można nie tylko czytać, ale również do niego pisać.
- Sposób komunikacji określony jest za pomocą parametru *mode* metody `open`.
- Dostępne tryby (m. in.):
  - `"r"` (domyślne) - tylko czytanie z pliku;
  - `"w"` - tylko pisanie do pliku, tworzy plik jeżeli nie istnieje;
  - `"a"` - pisanie do pliku, nowa zawartość dodawana jest na na końcu, jeżeli plik istnieje;
  - `"+"` - pisanie i czytanie jednocześnie (np. `"r+"`).

Importowanie

- Tworząc program będziemy korzystać z więcej niż jednego pliku lub modułu.
- Do zaciągania nowych funkcji i klas używamy wyrażenia `import`. Wyrażenie to pozwala dodać do przestrzeni nazw cały moduł lub tylko konkretną funkcję.

```
import csv  # dodaje obiekt modulu csv
from csv import reader  # zaciaga tylko metode reader
```

- Istnieje kilkanaście modułów wbudowanych (`os`, `sys`, `collections`, `csv`, `...`) dostępnych w podstawowej dystrybucji Pythona. Dodatkowe moduły należy doinstalować przez narzędzie `pip`.

Thanks!!