Python

Trzeci wykład

dr inż. Marcin Ciura, Uniwersytet Komisji Edukacji Narodowej

Przepraszam was za to, że dziś nie każdy przykład będzie po czesku :-)

Proszę robić notatki :-)

Plan na dziś

+ Trzy typy proste:
  - łańcuch (**str**)
  - liczba całkowita (**int**)
  - liczba rzeczywista (**float**)

+ f-łańcuchy
+ łańcuchy wieloliniowe

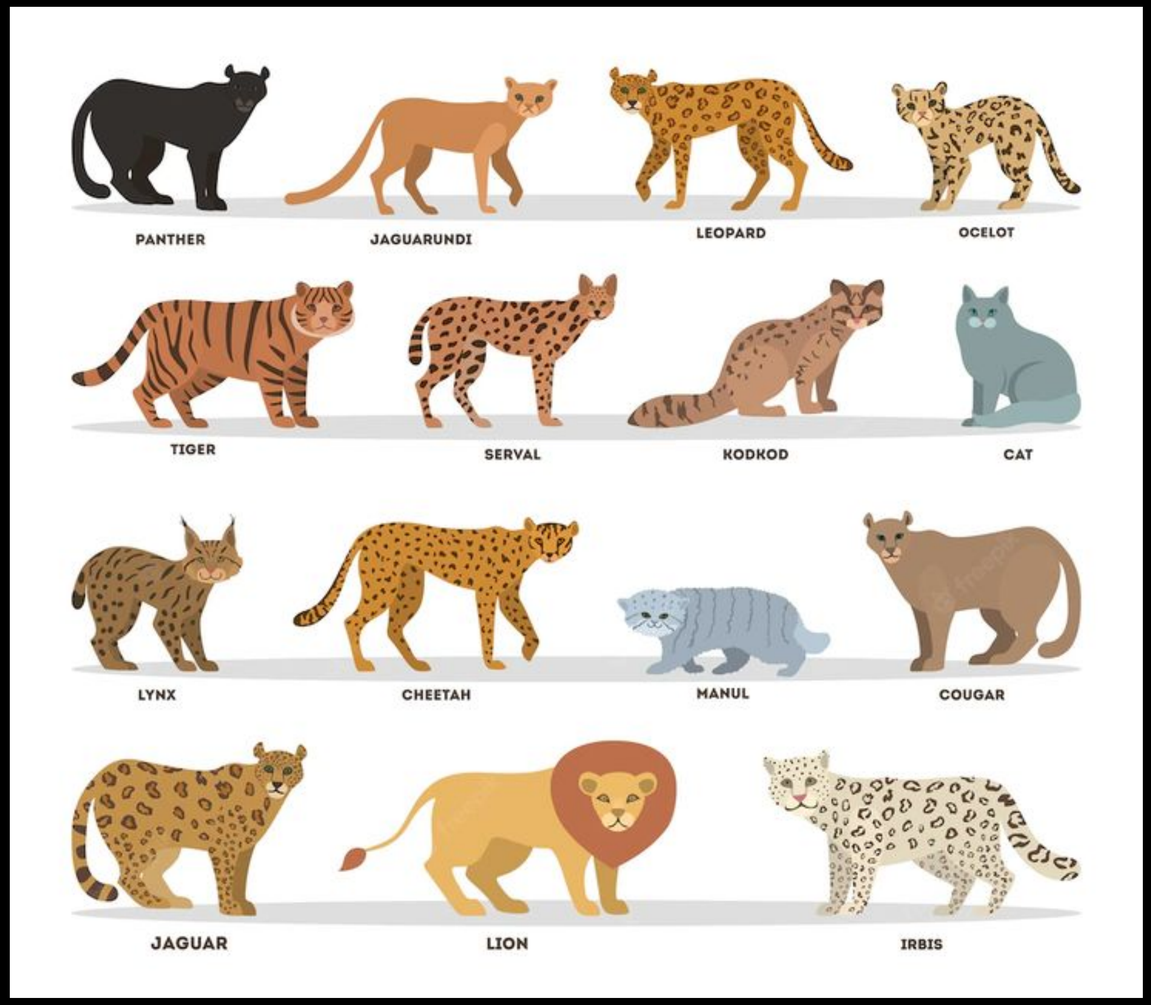
+ instrukcje **if**…**else** i **if**…**elif**…**else**

+ jak definiować własne funkcje?

+ i dwa typy złożone:
  - lista (**list**)
  - słownik (**dict**)

Zagadka na dobry początek :-)

Proszę zgadnać, jak się nazywają kotowate po czesku



Jak się nazywają kotowate po czesku?

Kočkovité šelmy :-)

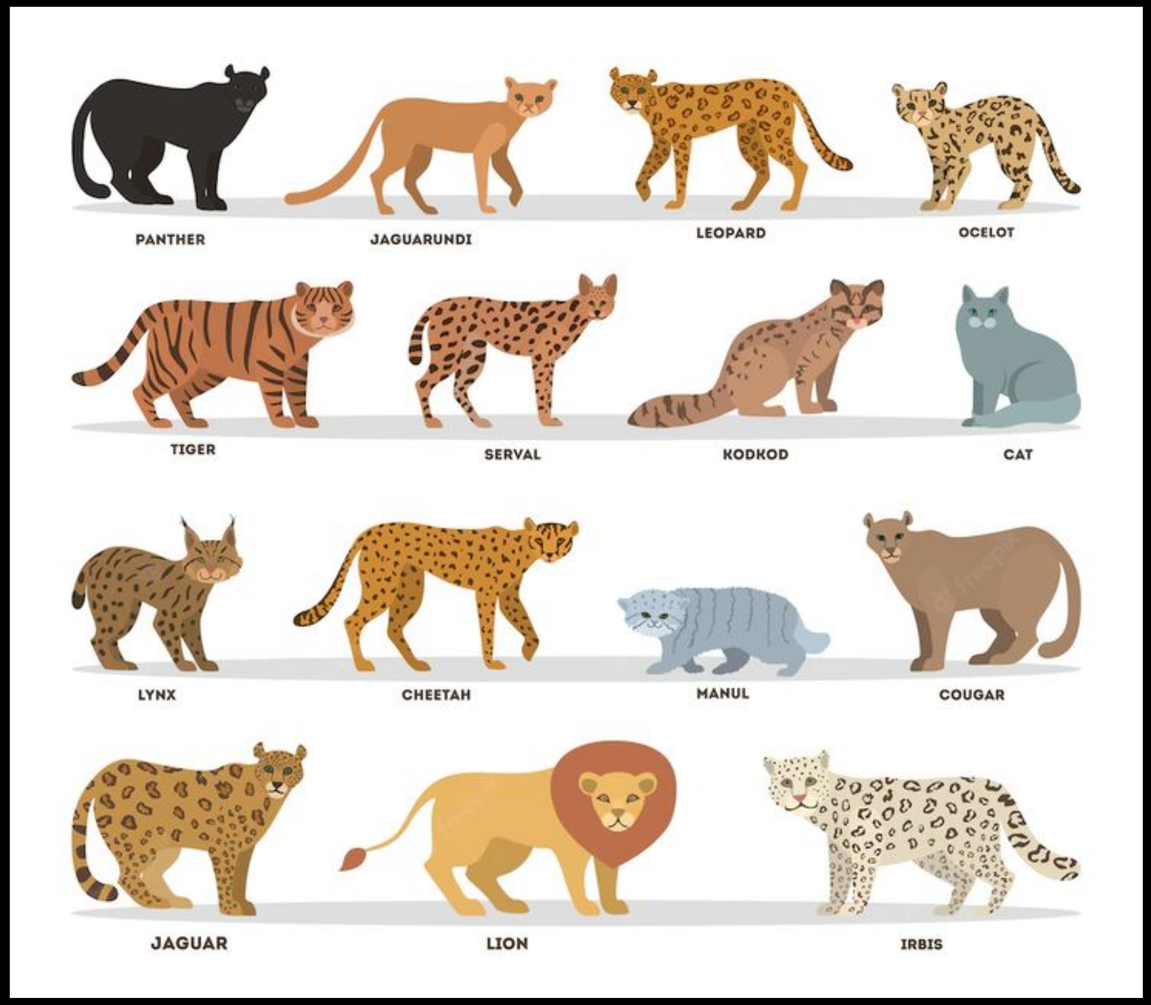
Tak czytamy litery z haczykami:

č ř š ž ň ě – cz, rz, sz, ż, ń, ie

Tak czytamy litery z akcentami:

á é í ó ú ů ý –

aa, ee, ii, oo, uu, óó, ii



Co to jest obiekt? Co to jest typ?

Pojęcie „obiekt danego typu”
jest podobne do pojęcia „osobnik danego gatunku”

Osobniki różnych gatunków:
+ mają różny wygląd
+ mają różne zwyczaje

Obiekty różnych typów:
+ przechowują wartości z różnych dziedzin
+ mają różne zestawy działań, które można na nich wykonywać

Typy danych

Typ **str**, czyli typ tekstowy

Zmienne typu **str** (**str** to skrót wyrazu „string”, czyli „sznurek”) przechowują teksty, czyli łańcuchy znaków

Typ **str**, czyli typ tekstowy — powtórka

Możemy dodać łańcuch do łańcucha za pomocą operatora +, na przykład:

```
imię = 'Jiří'
powitanie = 'Ahoj, ' + imię + '!'
print(powitanie)
```

```
Ahoj, Jiří!
```

Typ **str**, czyli typ tekstowy

Możemy pomnożyć łańcuch przez liczbę całkowitą, stawiając operator * z lewej lub z prawej strony łańcucha, na przykład:

```
print('xo' * 20)
xoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxo
print(6 * '#')
######
```

Typ **str**, czyli typ tekstowy — powtórka

Możemy porównać łańcuch z łańcuchem za pomocą operatorów == (równa się), != (nie równa się), na przykład:

```
print('jiří' == 'Jiří')
False
print('Jiří' != 'Jiží')
True
```

Typ **str**, czyli typ tekstowy

Możemy też porównać łańcuch z łańcuchem za pomocą operatorów < (mniejszy niż), <= (mniejszy lub równy), > (większy niż), >= (większy lub równy). Te operatory porównują kolejność alfabetyczną łańcuchów, na przykład:

```
print('gepard' < 'puma')
True
print('tygr' <= 'lev')
False
```

Typ **int**, czyli liczby całkowite

Zmienne typu **int** (**int** to skrót wyrazu „integer”, czyli „liczba całkowita”) przechowują liczby całkowite :-)

```
0  9  -23
```

W dużych liczbach całkowitych możemy dla czytelności wstawiać znaki podkreślenia _ (w małych liczbach też :-)

```
1_000_000 == 1_000 * 1_000
```

Typ **int**, czyli liczby całkowite

Na liczbach całkowitych możemy wykonywać te działania:
+ (dodawanie), - (odejmowanie), * (mnożenie),
/ (dzielenie), ** (potęgowanie)

```
print(2+2, 2-3, 2*2, 2/3, 2**3)
```

```
4  -1  4  0.6666666666666666  8
```

Typ **int**, czyli liczby całkowite

W Pythonie wynik dzielenia liczb całkowitych jest zawsze liczbą rzeczywistą

```
print(2/3, 8/2)
```

```
0.6666666666666666  4.0
```

Typ **float**, czyli liczby rzeczywiste

Zmienne typu **float** (**float** to skrót wyrażenia „floating-point number”, czyli „liczba zmiennoprzecinkowa”) przechowują liczby rzeczywiste

Liczby rzeczywiste odróżniamy od liczb całkowitych dzięki kropce dziesiętnej (Anglosasi używają kropek dziesiętnych zamiast przecinków dziesiętnych)

0.0 -1.0 2.5 0.14285714285714285

Działania na liczbach całkowitych i rzeczywistych

Działania +, -, *, /, ** możemy wykonywać i na liczbach całkowitych, i na liczbach rzeczywistych. Wystarczy jedna liczba rzeczywista, żeby wynik był rzeczywisty:

```
print(2 + 3*4.0)
print(2 + 3*4)
14.0
14
```

Działania na liczbach całkowitych i rzeczywistych

Możemy używać nawiasów. Wszystkie nawiasy powinny być okrągłe, nawet te zagnieżdżone

```
print((2 + 3)*4.0)
20.0
```

Działania na liczbach całkowitych i rzeczywistych

Działania ==, !=, <, <=, >, >= możemy wykonywać i na liczbach całkowitych, i na liczbach rzeczywistych:

```
print(2 == 2.0)
True
print(-7.5 < 0)
True
```

Działania na liczbach całkowitych i rzeczywistych

Liczby rzeczywiste przechowują mniej więcej 15 cyfr znaczących. Proszę się nie dziwić, gdy wyniki działań na liczbach rzeczywistych są niedokładne. Banki nie korzystają z typu **float** :-)

```
pierwiastek_z_2 = 2 ** 0.5
print(pierwiastek_z_2)
1.4142135623730951
print(pierwiastek_z_2 ** 2)
2.0000000000000004
```

Jak poznać typ zmiennej?

+ Możemy pamiętać, jakiego typu wartość przypisaliśmy do tej zmiennej

+ Możemy używać czytelnych nazw zmiennych: liczba_studentów, średnia_ocena

+ Możemy trzymać się dobrych tradycji: zmienne i, j, k całkowite, zmienne x, y, z rzeczywiste

Jak poznać typ zmiennej?

+ Możemy też wywołać funkcję wbudowaną **type**:

```
print(type(7))
<class 'int'>
print(type(7.0))
<class 'float'>
print(type('7'))
<class 'str'>
```

Konwersje typów

Czasem Python niejawnie konwertuje typy, na przykład wtedy, gdy łączymy w jednym wyrażeniu wartości typu **int** z wartościami typu **float**

Konwersje typów

Możemy też jawnie konwertować typy, wywołując funkcje wbudowane **str**, **int**, **float**:

```
dzień_miesiąca = 25
print('Dnes je ' + dzień_miesiąca + ' listopad')
TypeError: can only concatenate str (not "int") to str
print('Dzisiaj jest ' + str(dzień_miesiąca) + ' listopada')
Dzisiaj jest 25 listopada
```

Konwersje typów

Możemy też jawnie konwertować typy, wywołując funkcje wbudowane **str**, **int**, **float**:

str(7.5) == '7.5'

int('9') == 9

int(12.5) == 12

Konwersje typów

Kiedy nasz program wczytuje liczby funkcją **input**,
musimy skonwertować typ **str** na typ **int**

Konwersje typów

```
a, b = input('Podaj dwie liczby oddzielone spacją: ').split()
print(type(a), type(b))
a = float(a)
b = float(b)
print(a * b)
```

```
Podaj dwie liczby oddzielone spacją: 4 5
<class 'str'> <class 'str'>
20.0
```

Typy danych
Podsumowanie

Typy danych — podsumowanie

`str + str`, `str * int`, `int * str`

`int + int`, `int - int`, `int * int`, `int / int`, `int ** int`

`float + float`, `float - float`, `float * float`,
`float / float`, `float ** float`

`x == y`, `x != y`, `x < y`, `x <= y`, `x > y`, `x >= y`

`str(x)`, `int(x)`, `float(x)`

Proszę mi zadać pytanie :-)

f-łańcuchy

f-łańcuchy

f-łańcuchy, czyli formatowane łańcuchy, to fajny sposób tworzenia łańcuchów w Pythonie

Dzięki f-łańcuchom programy są bardziej czytelne, bardziej zwięzłe i bardziej wydajne

f-łańcuchy

Tworzymy f-łańcuch, poprzedzając literą f cudzysłów, który rozpoczyna łańcuch

Wewnątrz f-łańcucha umieszczamy w nawiasach klamrowych { } te wyrażenia, które chcemy wstawić do łańcucha, na przykład:

```
imię = 'Marcin'
nr_wykładu = 3
print(f'Mam na imię {imię}. To jest {nr_wykładu}. wykład.')
Mam na imię Marcin. To jest 3. wykład.
```

f-łańcuchy

Wewnątrz f-łańcucha umieszczamy w nawiasach klamrowych { } te wyrażenia, które chcemy wstawić do łańcucha. Mogą to być dowolne wyrażenia, nie tylko pojedyncze nazwy, na przykład:

```
a = 7
b = 4
print(f'Wynik działania {a} + {b} = {a + b}')
```

```
Wynik działania 7 + 4 = 11
```

f-łańcuchy

Wewnątrz f-łańcucha łatwo możemy podawać długość pola, czyli ile znaków ma zajmować łańcuch, i wyrównywać tekst do lewej lub do prawej strony pola, albo umieszczać go pośrodku

Służy do tego operator wyrównania. Operator wyrównania to:
+ > (wyrównaj w prawo)
+ < (wyrównaj w lewo)
+ albo ^ (wyśrodkuj)
Stawiamy go po dwukropku : wewnątrz nawiasów klamrowych { }

f-łańcuchy

Po operatorze wyrównania piszemy szerokość pola,
w którym chcemy umieścić łańcuch
Python uzupełni to pole spacjami

```
imię = 'Kleopatra'
print(f'|{imię:<20}|{imię:>20}|{imię:^20}|')
|Kleopatra           |           Kleopatra|    Kleopatra     |
```

f-łańcuchy

Po dwukropku : a przed operatorem wyrównania możemy podać znak wypełnienia. Gdy podamy znak wypełnienia, Python wypełni nim pole

```
n = 42
# Wypisz n uzupełnione zerami, wyrównane do prawej
# strony pola o długości 5 znaków
print(f'{n:0>5}')
```

```
00042
```

f-łańcuchy

```
tytuł = 'Zaklínač'
# Wypisz tytuł uzupełniony gwiazdkami, wyśrodkowany
# w polu o długości 20 znaków
print(f'{tytuł:*^20}')
*******Zaklínač*******
```

f-łańcuchy

f-łańcuchy ze znakiem wypełnienia > przydają się do równego wypisywania liczb w tabelach: cyfry jedności pod cyframi jedności, cyfry dziesiątek pod cyframi dziesiątek i tak dalej:

```
# Wypisz liczby wyrównane do prawej strony w polu o szerokości 10 znaków
liczba = 123
print(f'{liczba:>10}')
liczba = 4321
print(f'{liczba:>10}')
```

```
       123
      4321
```

f-łańcuchy

Możemy napisać po szerokości kropkę . i precyzję
Precyzja określa liczbę miejsc po kropce dziesiętnej
w liczbach rzeczywistych. Gdy wypisujemy liczby
rzeczywiste, radzę dopisać po precyzji literę f (jak **float**)

f-łańcuchy

```
# Wypisz pi wyrównane uzupełnione spacjami, wyrównane
# do prawej strony w polu o szerokości 10 znaków,
# z 2 cyframi po przecinku. Liczba pi to liczba
# rzeczywista (f)
pi = 3.14159265358979 3
print(f'{pi:>10.2f}')
      3.14
```

Skoro f-łańcuchy są takie fajne, to czemu wszystkie łańcuchy nie są f-łańcuchami?

Z powodów historycznych. Python się rozwija
Na początku były w nim tylko zwykłe łańcuchy
Dopiero później wprowadzono do niego f-łańcuchy
Dodano im literę f przed cudzysłowem, by odróżnić je od zwykłych łańcuchów

Gdyby f-łańcuchy nie odróżniały się od zwykłych łańcuchów, zmieniłoby się działanie tych starszych programów,
w których łańcuchy zawierają nawiasy klamrowe { },
Twórcy Pythona nie chcieli do tego dopuścić

Łańcuchy wieloliniowe

Łańcuchy wieloliniowe

Łańcuchy wieloliniowe to takie łańcuchy, które mają więcej niż jedną linię :-)

Do tworzenia łańcuchów wieloliniowych używamy potrójnych cudzysłowów podwójnych """ lub potrójnych cudzysłowów pojedynczych '''

Łańcuchy wieloliniowe

Do tworzenia łańcuchów wieloliniowych używamy potrójnych cudzysłowów """ lub ''', na przykład:

```
print("""Lokomotiva

Na dráze stojí lokomotiva,
paří se, olej s povrchu splývá,
ospale zívá.

Syčí a sípe, zhluboka dýchá,
pára jí dmýchá z horkého břicha:
""")  # Přeložil Jacek Baluch
```

Łańcuchy wieloliniowe

Łańcuchy wieloliniowe mogą być f-łańcuchami, na przykład:

```
list = f"""Drogi {imię},
Nie wiem, czy mnie pamiętasz. Mam na imię {moje_imię}.
Poznaliśmy się {jak_dawno} temu {gdzie}.
{treść}
Pozdrawiam,
{moje_imię}
"""
```

f-łańcuchy i łańcuchy wieloliniowe
Podsumowanie

f-łańcuchy i łańcuchy wieloliniowe — podsumowanie

```
print(f'Wynik działania {a} * {b} = {a * b}')

print(f'|{x:<10}|{x:>10}|{x:^10}|{x:>10.2f}')

print("""Koukejme, co vám je ve vagóně:
V prvním jsou krávy,
ve druhém koně""")
```

Proszę mi zadać pytanie :-)

Instrukcje if…else i if…elif…else

Instrukcja if…else

Instrukcja if…else zawiera dwa bloki instrukcji. Kod w bloku po części if zostanie wykonany, jeśli warunek jest prawdziwy (True). Kod w bloku po części else zostanie wykonany, jeśli warunek jest nieprawdziwe (False)

```
if wiek >= 18:
    print('Osoba dorosła')
else:
    print('Osoba niepełnoletnia')
```

Po co nam instrukcja if…elif…else

Instrukcja if…elif…else służy do wyboru spośród wielu możliwości. Nie wymaga dodatkowego wcięcia

Słowo kluczowe elif to skrót od "else if"

Po co nam instrukcja if…elif…else

Zamiast pisać tak:

```
if ocena == 5:
    print('bardzo dobry')
else:
    if ocena == 4:
        print('dobry')
    else:
        if ocena == 3:
            print('dostateczny')
        else:
            print('niedostateczny')
```

wygodniej jest pisać tak:

```
if ocena == 5:
    print('bardzo dobry')
elif ocena == 4:
    print('dobry')
elif ocena == 3:
    print('dostateczny')
else:
    print('niedostateczny')
```

Jak definiować własne funkcje?

Co to jest funkcja w językach programowania?

Funkcja to samodzielny fragment programu, który wykonuje pewne zadanie na określonych danych. Program może wiele razy wywoływać tę samą funkcję w różnych miejscach programu z różnymi danymi

Funkcje mają swoje nazwy
Funkcje mogą mieć swoje dane wejściowe. Mówimy na te dane argumenty funkcji
Funkcje mają swój kod, czyli instrukcje, które wykonuje komputer

Jak definiować własne funkcje?

Aby zdefiniować własną funkcję, piszemy po kolei:
+ słowo kluczowe **def** (od "define", czyli „zdefiniuj")
+ nazwę funkcji
+ nawiasy okrągłe ()
+ dwukropek :
+ kod funkcji, wcięty o kilka spacji

```
def powitanie():
    print('Ahoj :-)')
```

Jak definiować własne funkcje?

Dobry zwyczaj to wpisać na początku kodu funkcji tak zwany docstring
(od "documentation string"), czyli łańcuch, który opisuje, co robi ta funkcja. Zgodnie z tradycją otaczamy docstringi potrójnymi znakami podwójnego cudzysłowu """

```
def powitanie():
    """Wypisuje powitanie na ekranie"""
    print('Ahoj :-)')
```

Jak definiować własne funkcje?

```
def powitanie():
    """Wypisuje powitanie na ekranie"""
    print('Ahoj :-)')

# Tak wywołujemy funkcję powitanie. Zawsze wywołujemy
# funkcję, pisząc po jej nazwie nawiasy okrągłe ()
powitanie()
```

Jak definiować własne funkcje?

Dzięki docstringom i funkcji wbudowanej **help** możemy bez zaglądania do kodu dowiedzieć się, co robią funkcje, nawet te, których definicji nie pamiętamy

```
def powitanie():
    """Wypisuje powitanie na ekranie"""
    print('Witaj :-)')

help(powitanie)
```

```
Help on function powitanie in module __main__:

powitanie()
    Wypisuje powitanie na ekranie
```

Jak definiować własne funkcje?

Funkcje często potrzebują danych, żeby działać. Dane przekazujemy do funkcji jako argumenty w nawiasach okrągłych ( )

Dobry zwyczaj to pisać po każdym argumencie dwukropek : i typ tego argumentu

Jak definiować własne funkcje?

```
def powitaj_imiennie(imię: str):
    """Wypisuje na ekranie powitanie danej osoby

    Argumenty:
    imię - kogo powitać?
    """
    print(f'Ahoj, {imię}!')
```

Jak definiować własne funkcje?

```
def powitaj_w_języku(imię: str, język: str):
    """Wypisuje na ekranie powitanie w danym języku

    Argumenty:
    imię - kogo powitać?
    język - 'pl' lub 'cs'
    """
    if język == 'pl':
        print(f'Witaj, {imię}!')
    elif język == 'cs':
        print(f'Ahoj, {imię}!')
```

Jak definiować własne funkcje?

Wiele funkcji zwraca wynik swojego działania
Służy do tego słowo kluczowe **return** („zwróć")

Gdy funkcja napotyka słowo kluczowe **return**, od razu kończy działanie i przekazuje tę wartość, która stoi za słowem kluczowym **return** tam, skąd została wywołana
Dobry zwyczaj to pisać po nawiasie zamykającym argumenty strzałkę -> i typ wyniku

Jak definiować własne funkcje?

```
def stwórz_podpis(imię: str, nazwisko: str) -> str:
    """Zwraca połączone imię i nazwisko danej osoby

    Argumenty:
    imię - imię osoby
    nazwisko - nazwisko osoby
    """
    return f'{imię} {nazwisko}')
```

Jak definiować własne funkcje?

```
def średnia3(a: float, b: float, c: float) -> float:
    """Zwraca średnią trzech liczb

    Argumenty:
    a, b, c - liczby, których średnią trzeba obliczyć
    """

    return (a + b + c) / 3
```

Jak definiować własne funkcje?

```
def oblicz_procent(liczba: float, całość: float) -> float:
    """Oblicza, jaki procent całości stanowi liczba

    Zwraca 0.0, jeśli całość jest równa 0

    Argumenty:
    liczba, całość - liczby rzeczywiste
    """
    if całość == 0.0:
        return 0.0
    return 100.0 * (liczba / całość)
```

Jak definiować własne funkcje? Podsumowanie

Jak definiować własne funkcje? — podsumowanie

```
def nazwa_funkcji(argument_1: typ, argument_2: typ) -> typ:
    """Docstring: co robi nazwa_funkcji

    Argumenty:
    argument_1 - co to jest?
    argument_2 - co to jest?
    """
    # Tutaj funkcja coś robi
    return wynik
```

Proszę mi zadać pytanie :-)

Kiedy będzie ciąg dalszy „Lokomotywy”?

Czas na czeski dowcip o języku polskim

Jak se řekne polsky „Nástup do dvou řad“?
Jak powiedzieć po polsku „W dwuszeregu zbiórka“?

Czas na czeski dowcip o języku polskim

Jak se řekne polsky „Nástup do dvou řad“?
Jak powiedzieć po polsku „W dwuszeregu zbiórka“?

Proszę pana wejść za pana,
aby pan nie widział pana :-)

Listy
(chodzi o spisy rzeczy,
nie o kartki w kopertach :-)

Listy

Lista to przykład kolekcji obiektów. Dzięki kolekcjom możemy przechowywać wiele obiektów w jednym obiekcie

Listy:
+ są uporządkowane — każdy element ma swój indeks
+ można zmieniać, czyli można dodawać, usuwać lub zmieniać ich elementy
+ mogą zawierać dowolne elementy. W 99% przypadków elementy jednej listy są tego samego typu, bo inaczej jest nam niewygodnie

Listy

Przykłady list:

```
liczebniki = ['nula', 'jedna', 'dva', 'tři', 'čtyři', 'pět']
pusta_lista = []
rozbiory = [1772, 1793, 1795]
```

Indeksowanie list

Listy indeksuje się od zera, tak jak łańcuchy

```
print(liczebniki[0])
print(liczebniki[2])
print(liczebniki[-1])
```

```
nula
dva
pět
```

Indeksowanie list

Listy indeksuje się od zera, tak jak łańcuchy
Łańcuchów nie można zmieniać, a listy można zmieniać

```
# Do listy liczebniki pod indeksem 1 przypisz
# łańcuch 'JEDNA'
liczebniki[1] = 'JEDNA'
print(liczebniki)
liczebniki = ['nula', 'JEDNA', 'dva', 'tři', 'čtyři', 'pět']
```

Listy — metoda .append

Metoda .append dodaje jeden element na koniec listy

```
liczebniki.append('šest')
print(liczebniki)
['nula', 'jedna', 'dva', 'tři', 'čtyři', 'pět', 'šest']
```

Funkcja **len**

Funkcja **len** zwraca długość listy

```
print(len(rozbiory))
print(len(pusta_lista))
```

```
3
0
```

Funkcja **len**

Proszę pamiętać o tym, że ostatni element listy L ma indeks len(L) - 1

```
print(len(rozbiory))
print(rozbiory[2])
```

```
3
1795
```

Przeglądanie elementów listy (iterowanie po elementach listy)

Dzięki pętli **for** możemy przeglądać elementy listy

```
# Zmienna rok przyjmuje w kolejnych obiegach pętli
# wartości kolejnych elementów listy rozbiory
for rok in rozbiory:
    print(rok)
```

```
1772
1793
1795
```

Przeglądanie elementów listy (iterowanie po elementach listy)

```
for liczebnik in liczebniki:
    print(liczebnik)
```

```
nula
jedna
dva
tři
čtyři
pět
šest
```

Łańcuchy — metoda .split — uzupełnienie

Metoda .split dzieli łańcuch na listę jego podłańcuchów na podstawie podanego separatora
Jeśli nie podamy separatora, metoda .split dzieli łańcuch tam, gdzie są w nim białe znaki

```
print('Náhle svist! Náhle hvizd!'.split())
['Náhle', 'svist!', 'Náhle', 'hvizd!']
```

Łańcuchy — metoda .split — uzupełnienie

Jako separator możemy podać dowolny ciąg znaków

```
print("""Copak to, jakpak to uvádí v běh,
odkud se bere ten šílený spěch?
Jak to, že pádí, že bouchá buc-buch?""".split('\n'))
['Copak to, jakpak to uvádí v běh,', 'odkud se bere ten
šílený spěch?', 'Jak to, že pádí, že bouchá buc-buch?']
```

Łańcuchy — metoda .split — uzupełnienie

```
kočkovité = 'kočka,lev,tygr,gepard,puma,jaguár'.split(',')
print(kočkovité)
['kočka', 'lev', 'tygr', 'gepard', 'puma', 'jaguár']
```

Łańcuchy — metoda .join

Metoda .join działa odwrotnie do metody .split
Metoda .split dzieli, metoda .join łączy

Metoda .join łączy listę łańcuchów w łańcuch, wstawiając separator między kolejnymi elementami listy
Ważne: metodę .join wywołujemy na separatorze

Łańcuchy — metoda .join

```
kočkovité = ['kočka', 'lev', 'tygr', 'gepard', 'puma',
    'jaguár']
print(''.join(kočkovité))  # Separatorem jest łańcuch pusty
print(' '.join(kočkovité))  # Separatorem jest spacja
print('--'.join(kočkovité))
```

```
kočkalevtygrgepardpumajaguár
kočka lev tygr gepard puma jaguár
kočka--lev--tygr--gepard--puma--jaguár
```

Listy
Podsumowanie

Listy — podsumowanie

```
lista = [2, 3, 5, 7, 9]
lista[1] == 3
lista[-1] = 11
lista.append(13)
len(lista) == 5

for liczba in lista:
    print(liczba ** 2)
```

```
print('2,0,2'.split(','))
['2', '0', '2']

print('-'.join(['2', '0', '2']))
2-0-2
```

Proszę mi zadać pytanie :-)

Słowniki

Zadanie
Proszę podać tytuł jakiegoś słownika
Każda odpowiedź jest dobra :-)

Słowniki

Słowniki służą do tłumaczenia kluczy na wartości*
Słownik w Pythonie to kolekcja par: klucz i wartość
Dzięki słownikowi możemy wygodnie poznać tę wartość, która odpowiada danemu kluczowi

* Słowniki ortograficzne są inne. W Pythonie odpowiadają im zbiory, o których będzie mowa na innym wykładzie

Słowniki

Słowniki:
+ można zmieniać, czyli można dodawać, usuwać lub zmieniać pary klucz-wartość
+ w danym słowniku każdy klucz musi być inny
+ kluczami słowników mogą być łańcuchy lub liczby
+ wartościami słowników mogą być łańcuchy, liczby, listy, słowniki, a nawet funkcje

Słowniki

Zawartość słownika otaczamy nawiasami klamrowymi { }
Wartości oddzielamy od kluczy dwukropkami :
Pary klucz-wartość oddzielamy od siebie przecinkami ,
Po ostatniej parze klucz-wartość możemy postawić przecinek albo nie

Słowniki

```
ceny = {
    'jablka': 5.99,
    'banány': 6.99,
    'pomeranče': 8.99,
}
```

Słowniki

W 99% przypadków typ kluczy danego słownika jest taki sam
W 95% przypadków* typ wartości danego słownika jest taki sam

```
słownik_polsko_czeski = {
    'jeden': 'jedna',
    'dwa': 'dwa',
    'trzy': 'tři',
}
```

* Według badań naukowych 73,6% statystyk jest wyssanych z palca

Słowniki — operator indeksowania [ ]

Dzięki operatorowi indeksowania [ ] dowiadujemy się, jaka wartość odpowiada danemu kluczowi. Operator [ ] jest bardzo często używany

Słowniki

Tak odczytujemy wartość, która odpowiada kluczowi:

```
print(ceny['jablka'])
print(słownik_polsko_czeski['dwa'])
print(słownik_polsko_czeski['siedemnaście'])
```

```
5.99
dva
KeyError: 'siedemnaście'
```

Słowniki — operator indeksowania [ ]

```
ceny = {
    'jablka': 5.99,
    'banány': 6.99,
    'pomeranče': 8.99,
}
owoce = 'jablka'
print(owoce, ceny[owoce])
```

```
jablka 5.99
```

Zasada DRY

Na następnym slajdzie zobaczymy dwa sposoby rozwiązania tego samego problemu. I sposób po lewej stronie, i sposób po prawej stronie jest prosty, ale sposób po prawej stronie jest bardziej elegancki. Sposób po prawej stronie jest zgodny z zasadą DRY:

Don't Repeat Yourself (Nie Powtarzaj Się)

Po co nam słowniki?

Zamiast pisać tak:

```
if język == 'pl':
    powitanie = 'Witaj'
elif język == 'cs':
    powitanie = 'Ahoj'
elif język == 'en':
    powitanie = 'Hello'
```

Można pisać tak:

```
powitanie = {
    'pl': 'Witaj',
    'cs': 'Ahoj',
    'en': 'Hello',
}[język]
```

Słowniki — operator in

Dzięki operatorowi in sprawdzamy, czy klucz jest w słowniku. Operator in jest często używany

```
if owoce in ceny:
    print(owoce, ceny[owoce])
else:
    print(f'{owoce} nemají cenu')
```

Słowniki — operator not in

Dzięki operatorowi not in sprawdzamy, czy dany klucz nie należy do słownika

```
if owoce not in ceny:
    print(f'{owoce}? Neznám takové ovoce')
```

Słowniki — operator indeksowania [ ] — ciąg dalszy

Dzięki operatorowi indeksowania [ ] zmieniamy tę wartość, która odpowiada danemu kluczowi:

```
ceny['jablka'] = 1.99
```

Słowniki

Tak dodajemy do słownika parę klucz-wartość:

```
ceny['hrušky'] = 6.49
słownik_polsko_czeski['siedem'] = 'sedm'
```

Słowniki

Tak iterujemy po kluczach słownika
W kolejnych obiegach pętli do zmiennej owoce są przypisywane kolejne klucze słownika ceny

```
for owoce in ceny:
    print(owoce, ceny[owoce])
```

```
jablka 1.99
banány 6.99
pomeranče 8.99
hrušky 6.49
```

Słowniki — metoda .items

Tak iterujemy po parach klucz-wartość słownika
W kolejnych obiegach pętli do pary zmiennych owoce, cena są przypisywane kolejne pary klucz-wartość słownika ceny

```
for owoce, cena in ceny.items():
    print(owoce, cena)
```

```
jablka 1.99
banány 6.99
pomeranče 8.99
hrušky 6.49
```

Słowniki — metoda .values

Tak iterujemy po wartościach słownika
W kolejnych obiegach pętli do zmiennej cena są przypisywane kolejne wartości ze słownika ceny

```
for cena in ceny.values():
    print(cena)
```

```
1.99
6.99
8.99
6.49
```

Słowniki
Podsumowanie

Słowniki — podsumowanie

```
liczby = {
    'nula': 0, 'jedna': 1}

print(liczby['nula'])
liczby['dva'] = 2

if 'tři' in liczby:
    print('mám tři')
```

```
if 'tři' not in liczby:
    print('nemám tři')

for k in liczby:
    print(k)
for k, w in liczby.items():
    print(k, w)
for w in liczby.values():
    print(w)
```

To wszystko na dziś
Dziękuję wam za uwagę :-)

Proszę mi zadać pytanie :-)

Czy ktoś z was czegokolwiek
nie zrozumiał z tego wykładu?

Do zobaczenia
na następnym wykładzie,
na którym poznamy kilka sztuczek

Źródła

https://stock.adobe.com/pl/images/wild-and-dometic-cats-set-collection-of-cat-family/223014183

https://commons.wikimedia.org/wiki/File:2014_Praga,_Hradczany,_odprawa_warty.JPG

Jacek Baluch, Jedzie po czesku Lokomotywa https://ruj.uj.edu.pl/server/api/core/bitstreams/8dbc44de-9a4f-41c4-89dd-79b8a1d80613/content