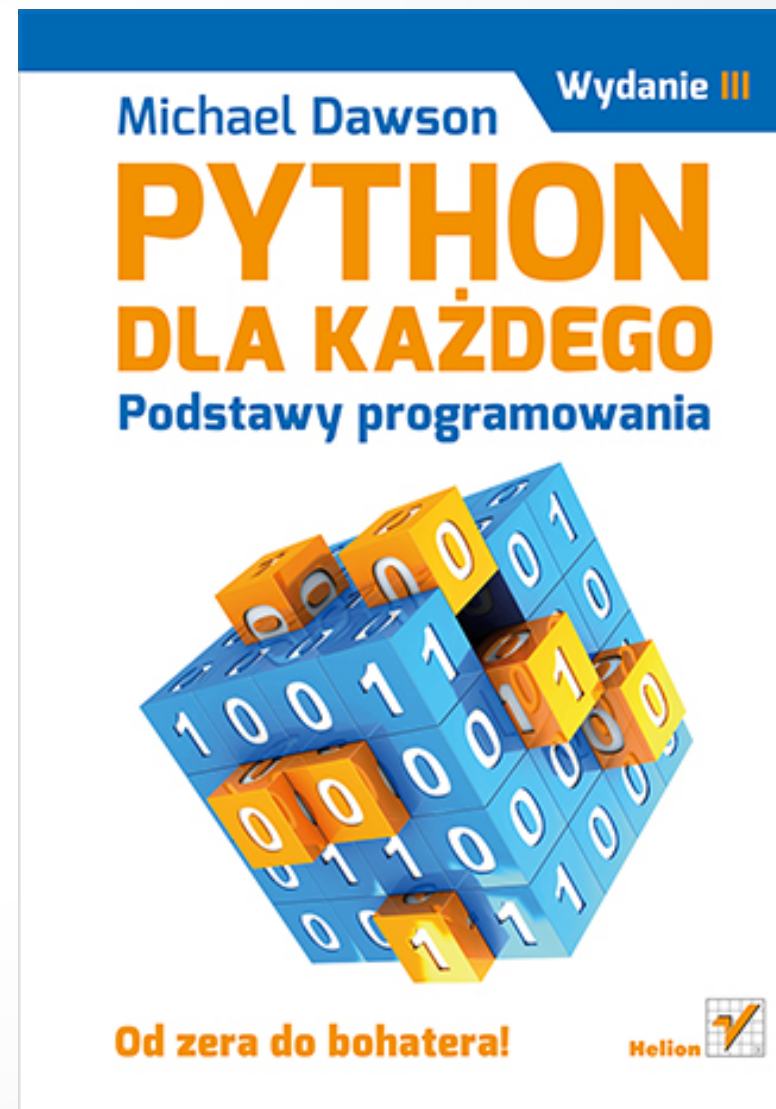


Wprowadzenie do Pythona

Literatura

Michael Dawson
„Python dla każdego“
Helion, 2014



Literatura

Mark Lutz

„Python, wprowadzenie“

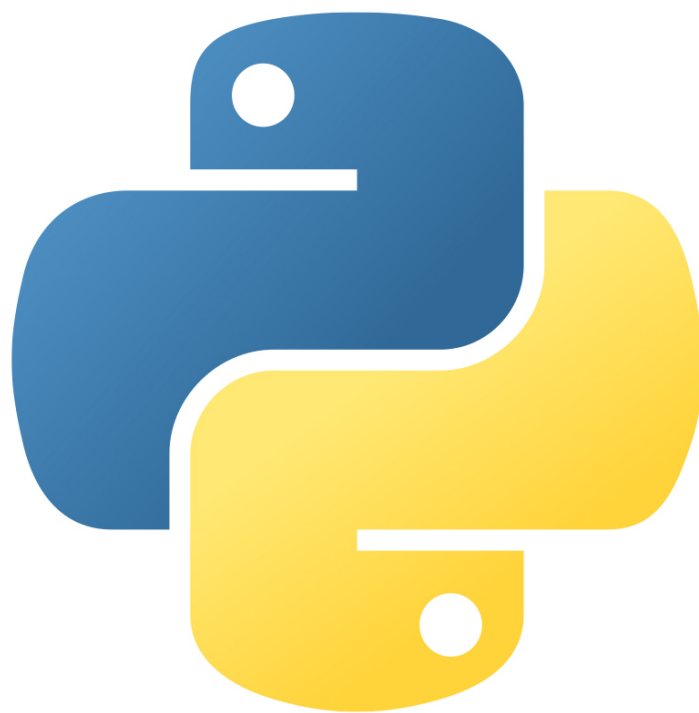
Helion, 2011



Python: skąd, jak i kto

- **Python** to nie gad – to Monty Python z Latającego Cyrku
- Guido van Rossum (ur. 1960, Holender) – stworzył Pythona w grudniu 1989, jak sam twierdzi, właściwie z nudów
- w roku 2005 Guido został zatrudniony w Google, a w 2013 przeniósł się do Dropboksa.

Python: jak wygląda?



Python: jaki jest?

- łatwy w nauce i użytkowaniu, intuicyjny, o dużej sile ekspresji
- publikowany jako *open source*
- do zastosowań ogólnych, ale użyteczny również w zastosowaniach niszowych
- skraca czas programowania i podnosi komfort pracy programisty

Python: niejedno ma imię

- w chwili obecnej w użyciu są dwie niezależne i niekompatybilne wersje Pythona:
- **Python 2** – podtrzymywany przy życiu ze względu na ogromną liczbę programów w nim napisanych (obecna wersja to 2.7)
- **Python 3** – stosowany do nowych projektów i traktowany jako przyszłościowy (obecna wersja to 3.5)

Pierwszy program

```
#!/usr/bin/python3
```

```
print("Hello world!")
```

Python: co nam daje?

- interpreter
- biblioteki
- proste (ale w pełni użyteczne) środowisko programisty o nazwie IDLE

Python:

- dwa sposoby pracy:
 - **interakcyjny** → w konsoli Pythona
 - **nieinterakcyjny** → „wsadowy“

składnia – znak szczególny Pythona:

- **jedna** instrukcja w jednym **wierszu** pliku
- instrukcji nie kończy się w żaden specjalny sposób
- można wstawiać do kodu puste linie, jeśli poprawi to czytelność kodu
- instrukcję można **złamać**, umieszczając na końcu linii znak \ (*backslash*) i kontynuując ją w następnym wierszu

```
print("Hello")
```

```
print \  
(\  
"world"\  
)
```


ważne:

- jak większość współczesnych języków programowania, **Python rozróżnia wielkość liter**

Komentarze

komentarz:

- **komentarz** zaczyna się tam, gdzie w linii stoi znak **#** (hash), a kończy się tam, gdzie kończy się ta linia
- nie ma komentarzy wielowierszowych...
- ...ale można w ich roli wykorzystać łańcuchy wielowierszowe

Literały

literały całkowite:

- 3000000000
- +-+-+123
- 0o10 # ósemkowo
- 0x10 # szesnastkowo
- 0b10 # dwójkowo

Konwersja do postaci znakowej

funkcja bin():

- `bin(x)`:
 - argument: liczba całkowita
 - wynik: postać binarna argumentu
 - efekt: żaden
- `np`:
- `print(bin(15))` → `0b1111`

Konwersja do postaci znakowej

funkcja hex():

- `hex(x)`:
 - argument: liczba całkowita
 - wynik: postać szesnastkowa argumentu
 - efekt: żaden
- np:
- `print(hex(15))` → `0xf`

Literały

literały rzeczywiste (zmiennopozycyjne):

- 2.49
- 1.24E-2

Literały

literały zespolone:

- $\text{real} + \text{realj}$
- $\text{real} - \text{realj}$

Literały

literały logiczne:

- prawda → True
- fałsz → False

Operator

literały arytmetyczne:

- $3 + 7.5$
- $3 - 7.5$
- $3 * 7.5$
- $3 / 7.5$
- $10 // 4$
- $10 \% 3$
- $2 ** 8$

Operator

literały bitowe (skrótowe!):

- $\sim x$
- $x \& y$
- $x | y$
- $x \wedge y$
- $x \ll y$
- $x \gg y$

Operator

literały relacyjne:

- $x > y$
- $x \geq y$
- $x < y$
- $x \leq y$
- $x == y$
- $x != y$

Operator

literały logiczne:

- **not** x
- **x and y**
- **x or y**

Ciekawostka

- osobliwością Pythona jest możliwość łączenia kolejno po sobie następujących wyrażeń logicznych łączonych spójnikiem `and`, dzięki czemu zamiast:

`x > 2 and x < 3`

można napisać:

`2 < x < 3`

operatory skrócone:

- w Pythonie funkcjonuje stary dobry pomysł zawierający się w konwencji mówiącej, że:

$$A = A \text{ OP } B$$

może zostać zapisane jako:

$$A \text{ OP} = B$$

czyli:

• $A = A + B$

→ $A += B$

• $A = A - B$

→ $A -= B$

• $A = A * B$

→ $A *= B$

• $A = A / B$

→ $A /= B$

• $A = A // B$

→ $A //= B$

• $A = A ** B$

→ $A ** = B$

oraz:

- $A = A \ll B$
- $A = A \gg B$
- $A = A \& B$
- $A = A | B$
- $A = A \wedge B$

- $A \ll= B$
- $A \gg= B$
- $A \&= B$
- $A |= B$
- $A \wedge= B$

ale nie mają postaci skróconej operatory
poniższe:

- $A = A \text{ and } B$
- $A = A \text{ or } B$

Zmienne

nazwa zmiennej:

- może zawierać litery (małe i wielkie), cyfry i znak _ (podkreślenie)
- nie może zaczynać się od cyfry
- może zawierać znaki narodowe
- litery małe i wielkie traktowane są jako różne

- **słowa kluczowe Pythona 3:**

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

operator przypisania =

- **efekt:** przypisanie wartości wyrażenia z prawej strony operatora do zmiennej wymienionej po lewej stronie
- **wynik:** wartość wyrażenia z prawej strony

oznacza to, że przypisanie:

$$a = b = c = d = 1$$

należy rozumieć jako ciąg przypisań:

$$a = (b = (c = (d = 1)))$$

- ponadto, w Pythonie działa również coś takiego:

```
a = 1
```

```
b = 2
```

```
a, b = b, a
```

Elementarne wejście

funkcja input() - wariant #1

- `input()`:
 - argument: żaden
 - efekt: wczytanie wiersza danych z stdin
 - wynik: łańcuch

funkcja input() - wariant #2

- `input(x)`:
 - argument: podpowieź dla użytkownika
 - efekt: wypisanie podpowiezi i wczytanie wiersza
 - wynik: napis
- np:
`tekst = input("Przemów do mnie!")`

funkcja int()

- konwertuje tekst na liczbę całkowitą
- `int(x)`:
 - argument: tekst reprezentujący liczbę
 - efekt: konwersja tekstu do liczby całkowitej
 - wynik: skonwertowana liczba
 - uwaga: wyrzuca wyjątek w razie niepowodzenia
- np:
`ile = int(input())`

funkcja float()

- konwertuje tekst na liczbę rzeczywistą
- `float(x)`:
 - argument: tekst reprezentujący liczbę
 - efekt: konwersja tekstu do liczby rzeczywistej
 - wynik: skonwertowana liczba
 - uwaga: wyrzuca wyjątek w razie niepowodzenia
- np:
`cena = float(input())`

importowanie – sposób pierwszy

```
import math
```

efekty:

- wszystkie udogodnienia z modułu `math` stają się dostępne, ale..
- należy je identyfikować tzw. **nazwą kwalifikowaną**
np.:
`y = math.sqrt(x)`

importowanie – sposób drugi

```
from math import sqrt
```

efekty:

- tylko jawnie wymienione udogodnienia z modułu `math` stają się dostępne, ale..
- nie trzeba ich identyfikować nazwą kwalifikowaną
`x = sqrt(y)`

Struktury sterowania

instrukcja **if** – wariant podstawowy

if *warunek* :

warunek:

- wyrażenie dowolnego typu
dowolna wartość różna od zera lub niepusty łańcuch
traktowany jest jak **True**, pozostałe wartości jako
False

- to, które instrukcje stanowią **treść** instrukcji **if**, w Pythonie jest oznaczane **poziomem wcięciem**!
- wcięcie można uzyskać **spacjami** bądź **tabulacją**
- ten drugi wariant jest **zalecany**
- mieszanie obu wariantów jest **ryzykowne**
- powrót do poprzedniego poziomu wcięcia oznacza **koniec instrukcji if**

Na przykład:

```
x = float(input())  
if x >= 0.0:  
    s = sqrt(x)  
    print(s)
```

Struktury sterowania

- jeśli w ramach instrukcji if (i pozostałych „blokowych) wykonywana jest tylko jedna instrukcja, zapis można skrócić do:

if *warunek* : *instrukcja*

instrukcja **if** – wariant drugi

```
if warunek :  
    kod1  
else :  
    kod2
```

- jeśli warunek będzie równy **True**, instrukcja **if** uzna, że należy **wykonać** instrukcje stojące za nią
- w przeciwnym wypadku zostaną wykonane instrukcje za **else**

instrukcja **if** – wariant trzeci

```
if warunek1 :  
    kod1  
elif warunek2 :  
    kod2  
else :  
    kod3
```

instrukcja **if** – uwagi

- fraza **elif** może wystąpić **wielokrotnie** – może również nie wystąpić w ogóle
- fraza **else** może wystąpić co najwyżej jednokrotnie i musi być **ostatnia**
- żadna z tych fraz nie może wystąpić bez wcześniejszego wystąpienia frazy **if**

instrukcja **while** – postać pierwsza

```
while warunek1 :  
    kod1
```

instrukcja **while** – postać druga

```
while warunek1 :  
    kod1  
else:  
    kod2
```

- tak długo, jak *warunek1* będzie równy True, instrukcja while będzie wykonywać *kod1* (być może nie wykona go ani razu)
- jeśli *warunek1* będzie równy False, wykona się *kod2* - conajmniej raz

- jeżeli jedna instrukcja if/while/... jest **zawarta w innej** instrukcji if/while/..., to manifestuje się to **wzrastającym** zagłębieniem (wcięciem)
- bądź uważny, kiedy posługujesz się wcięciami – złe wcięcia zaowocują **złym zachowaniem kodu**
- **błędy tego rodzaju odszukuje się długo i żmudnie**

Elementarne wyjście

- sekwencja:

```
print("Ala")  
print("ma")  
print("kota")
```

wyprowadzi na ekran:

```
Ala  
ma  
kota
```

sekwencja:

```
print("Ala", end=" ")  
print("ma", end=" ")  
print("kota")
```

wyprowadzi na ekran:

Ala ma kota

a sekwencja:

```
print("Ala", end="")  
print("ma", end="")  
print("kota")
```

wyprowadzi na ekran:

Alamakota

sterowanie wykonaniem pętli:

- `break`
jeśli chcesz opuścić pętlę wcześniej
- `continue`
jeśli chcesz wcześniej rozpocząć następny obrót pętli

pętla **for** – pierwsza postać:

- `for x in range(min,max):`
 kod

- uwaga na uproszczenie – formalnie pętla `for` wymaga listy (tablicy) jako źródła wartości dla zmiennej sterującej
- można więc rzec, że w Pythonie pętla `for` jest *de facto* pętlą `foreach`

`range(0, max)`

można zapisać krócej jako:

`range(max)`

`range(x,y)`

generuje listę wartości

$x, x+1, x+2, \dots, y-2, y-1$

pętla **for** – druga postać:

- `for x in range(min,max):`

`kod1`

`else:`

`kod2`

- *`kod2`* wykona się, gdy wartości za frazę `in` ulegną wyczerpaniu

```
for x in reversed(range(min,max)):
```



funkcja odwracająca otrzymany
zakres (listę)

Listy (aka tablice)

```
# przypisanie listy  
lista = [1,3,7,15]
```


pusta = []

ta lista jest pusta – nie zawiera żadnych danych

zera = [0] * 10

ta lista zawiera dziesięć zer; zauważ, że tym kontekście operator * nie oznacza mnożenia, a **powielanie**

listy:

- indeksy list zaczynają się od **0**
- odwołanie się do nieistniejącego elementu listy kończy się wyrzuceniem wyjątku
- indeksowanie liczbami ujemnymi działa jak w Perlu (indeksowanie od końca), ale z zastrzeżeniem punktu poprzedniego

- zachowanie listy jako argumentu funkcji **print()**

```
lista = [0, 3, 12, 8, 2]  
print(lista)
```

```
[0, 3, 12, 8, 2]
```

operator **in**:

wartość **in** lista

odpowiada na pytanie:

czy wartość **występuje** na liście?

(True - False)

operator **not in**:

wartość **not in** lista

odpowiada na pytanie:

czy wartość **nie występuje** na liście?

(True - False)

wycinek listy:

`lista[ skąd : dokąd ]`

- tworzy **nową listę** z wycinka **istniejącej** listy
- uwaga – mamy tu tę samą pułapkę, co w funkcji `range()`
- możemy używać indeksów ujemnych

zapamiętaj:

`lista[ : dokąd ]`

jest równoważne

`lista[ 0 : dokąd ]`

funkcja len():

jak length → długość

- ustala długość listy (liczbę jej elementów)
- `len(x)`:
 - argument: pewna lista
 - wynik: długość tej listy
 - efekt: żaden
- np:
`print(len([ ])) → 0`

zapamiętaj:

`lista[ skąd : ]`

jest równoważne

`lista[ skąd : len(lista) ]`

zapamiętaj:

`lista[ : ]`

jest równoważne

`lista[ 0 : len(lista) ]`

```
nowalista = lista[ : ]
```

jest czymś zupełnie innym niż:

```
nowalista = lista
```

- to pierwsze tworzy nową listę
- to drugie przypisuje referencję do listy istniejącej

instrukcja (nie funkcja!) **del**:

- usuwa element lub wycinek z listy (o ile istnieje)
- `del L[x]`:
 - argument: element x listy L
 - efekt: usunięcie elementu/elementów
- np:
`del lista[0]`

zapamiętaj:

```
del lista[:]
```

usuwa **całą zawartość** listy
(usuwa to, co jest w kontenerze)

```
del lista
```

usuwa **listę** jako taką
(usuwa cały kontener)

dodanie elementu na koniec listy:

`lista.append(element)`

`lista.insert(gdzie, element)`

- powoduje dodanie elementu `element` na pozycję `gdzie` listy `lista`; pozostałe elementy zostają „rozepchnięte”

Napisy

imię = 'Nabuchodonozor'
nazwisko = "Kowalski"

funkcja len() ukazuje swoją drugą twarz:

```
x = 'ala ma kota'  
print(len(x))
```

jeśli potrzebujesz apostrofu w napisie ograniczonym cudzysłowami, to nie ma problemu
(uwaga – slajd zawiera lokowanie produktu)

```
y = "Idę do McDonald's"  
print(y)
```

Idę do McDonald's

jeśli potrzebujesz cudzysłowu w napisie ograniczonym apostrofami, to też nie ma problemu

```
z = 'Lubię "Latający cyrk Monty Pythona"'
print(z)
```

```
Lubię "Latający cyrk Monty Pythona"
```

cytowanie apostrofu:

```
y = 'Idę do McDonald\'s'  
print(y)
```

Idę do McDonald's

cytowanie cudzysłowu:

```
z = "Lubię \"Latający cyrk Monty Pythona\""  
print(z)
```

Lubię "Latający cyrk Monty Pythona"

napis wielowierszowy:

```
z = """Co to za zwierzę,  
powiedz mi proszę,  
róg ma na nosie,  
nogi ma w rosie."""  
print(z)
```

```
Co to za zwierzę,  
powiedz mi proszę,  
róg ma na nosie,  
nogi ma w rosie.
```

napis wielowierszowy – tak też można:

```
z = '''Co to za zwierzę,  
trudno mi orzec:  
czy nosorożec,  
czy rosonożec.'''  
print(z)
```

```
Co to za zwierzę,  
trudno mi orzec:  
czy nosorożec,  
czy rosonożec.
```

cytowanie cytowania

```
napis = "Ten znak \\ to backslash"  
print(napis)
```

```
Ten znak \ to backslash
```


operator konkatencji: + (plus)

- łączy lewy i prawy argument w jeden nowy napis
- np:
 $\text{'py'} + \text{'thon'} \rightarrow \text{'python'}$
 $\text{'thon'} + \text{'py'} \rightarrow \text{'thonpy'}$

operator powielania: * (gwiazdka)

- powiela jeden argument tyle razy, ile określono w drugim argumencie, dając w efekcie nowy napis

- np:

'py' * 2 → 'pypy'
10 * '*' → '*****'

funkcja ord(x):

jak ordinal → porządkowy

- zamienia znak na odpowiadający mu kod
- ord(x):
 - argument: pewien znak
 - wynik: jego kod
 - efekt: żaden
- np:
`print(ord('A')) → 65`

funkcja chr(x):

jak character → *znak*

- zamienia kod na odpowiadający mu znak
- chr(x):
 - argument: kod pewnego znaku
 - wynik: ten znak
 - efekt: żaden
- np:
`print(chr(65))` → A

• **Sequence types czyli sekwencje**

- typy danych służące do kolekcjonowania danych w **uporządkowany** sposób noszą w Pythonie angielską nazwę „sequence types” czyli **sekwencje**
- sekwencjami są **listy**
- sekwencjami są także **łańcuchy (napisy)**
- wszystkie sekwencje zachowują się w podobny sposób, mają wiele **wspólnych** funkcji i metod

- **Dane „immutable” (niezmienne)**
 - niektóre dane dostępne w Pythonie są traktowane jako „**immutable**” (niezmienialne)
 - oznacza to, że **nie wolno zmieniać ich zawartości**
 - łańcuchy w Pythonie są *immutable*
 - można jednak dowolnie tworzyć **nowe wartości** bazując na **wartościach już dostępnych**

Dane „immutable” (niezmienne)

- fakt, że łańcuchy w Pythonie są *niezmienne* pociąga za sobą kilka konsekwencji:

Po pierwsze:

- instrukcja `del` nie działa na znaki w napisach – jej użycie jest w tym kontekście niedozwolone!
- można jednak wykonać ją w odniesieniu do całego napisu

- **Po drugie:**

łańcuchy nie mają metody `append()`

- **Po trzecie:**

łańcuchy nie mają metody insert()

- Ale wszystko, co wiemy o wycinkach, ma również zastosowanie dla łańcuchów:

```
napis = "abdefg"  
print(napis[1])  
print(napis[-1])  
print(napis[1:3])  
print(napis[3:])  
print(napis[:3])
```

```
b  
g  
bd  
efg  
abd
```

- wykorzystamy tę okazję, aby pokazać wycinek z krokiem:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print(napis[0:10:1])  
print(napis[::1])  
print(napis[1:18:2])  
print(napis[:-1:3])  
print(napis[1:-1:4])
```

```
abcdefghij  
abcdefghijklmnopqrstuvwxyz  
bdfhjlnpr  
adgjmpsvy  
bfjnrv
```

Operator **in** działa tak, jak tego oczekujemy:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print('f' in napis)  
print('F' in napis)  
print('1' in napis)
```

```
True  
False  
False
```

Operator **not in** również:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print('f' not in napis)  
print('F' not in napis)  
print('1' not in napis)
```

```
False  
True  
True
```

- Lewym argumentem operatorów **in** i **not in** mogą być również napisy:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print('ghi' not in napis)  
print('ghi' in napis)  
print('Xyz' not in napis)  
print('xyz' in napis)
```

```
False  
True  
True  
True
```

funkcja min(sekw):

jak minimum

- podaje najmniejszy element sekwencji
- `min(sekw)`:
 - argument: pewna sekwencja
 - wynik: najmniejszy element w sekwencji
- np:
`print(min('aAbByYzZ')) → A`

funkcja max(sekw):

jak maximum

- podaje największy element sekwencji
- `max(sekw)`:
 - argument: pewna sekwencja
 - wynik: największy element w sekwencji
- np:
`print(max('aAbByYzZ')) → z`

metoda `index(elem)`:

- podaje położenie elementu w sekwencji
- `sekw.index(elem)`:
 - argument: pewien element
 - wynik: położenie elementu w sekwencji
- np:
`print('aAbByYzZ'.index('b')) → 2`

przykłady działania metody `index()`:

```
t = 'Ala ma kota'  
print(t.index('t'))  
print(t.index('A'))  
print(t.index('a'))
```

```
9  
0  
2
```

Uwaga: próba znalezienia położenia elementu, którego w sekwencji nie ma, skończy się wyrzuceniem wyjątku

```
t = 'Ala ma kota'  
i = t.index('z')
```

```
Traceback (most recent call last):  
  File "x.py", line 2, in <module>  
    i = t.index('z')  
ValueError: substring not found
```

metoda count(elem):

jak count → liczyć

- zlicza wystąpienie elementu w sekwencji
- `sekw.count(elem)`:
 - argument: pewien element
 - wynik: liczba wystąpień elementu w sekwencji
- np:
`print('abccabc'.count('b')) → 2`

Metody i funkcje łańcuchowe (stricte!)

metoda center(szer):

- zwraca kopię łańcucha wycentrowanego w polu o szerokości szer
- `str.center(szer):`
 - argument: szerokość docelowego pola
 - wynik: kopia str dopasowana do pola o podanej szerokości
- np:
`print('['+'ala'.center(5)+''])` → `[ ala ]`

metoda `center(szer,wypełn)`:

- działa jak `center`, ale zamiast spacji używa wypełniacza `wypełn`
- `str.center(szer,wypełn)`:
 - argumenty: `szer`: szerokość docelowego pola
`wypełn`: wypełniacz (dokładnie jeden znak!)
 - wynik: kopia `str` dopasowana do pola o podanej szerokości z użyciem wypełniacza `wypełn`
- `np`:
`print('ala'.center(5, '*'))` → `*ala*`

metoda endswith(przyrostek):

- sprawdza, czy łańcuch kończy się ciągiem znaków przyrostek
- `str.endswith(przyrostek):`
 - argumenty: przyrostek: łańcuch
 - wynik: True / False
- np:
`print('Ala ma kota'.endswith('ta')) → True`

metoda `find`(podłańcuch):

- szuka podłańcucha w łańcuchu (podobnie jak `index()`, ale bezpieczniej)
- `str.find(podłańcuch)`:
 - argumenty: podłańcuch: szukany łańcuch
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('Ala ma kota'.find('ma'))` → 4

Uwaga:

nie używaj `find()`, jeśli tylko chcesz sprawdzić, czy pewien znak **występuje w łańcuchu** – operator `in` będzie szybszy!

metoda `find(podłańcuch, start)`:

- szuka podłańcucha w łańcuchu od wskazanej pozycji
- `str.find(podłańcuch, start)`:
 - argumenty: podłańcuch: łańcuch
start: pozycja, od której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('Ala ma kota'.find('a', 4))` → 5

metoda `find(podłańcuch,start,meta)`:

- szuka podłańcucha w łańcuchu od pierwszej wskazanej pozycji, ale nie dalej niż druga wskazana pozycja (jak w wycinkach)
- `str.find(podłańcuch, start, meta)`:
 - argumenty: podłańcuch: łańcuch
start: pozycja, od której należy szukać
koniec: pozycja, do której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('Ala ma kota'.find('a',4,5)) → -1`

metoda `isalnum()`:

- sprawdza, czy łańcuch zawiera wyłącznie litery i cyfry
- `str.isalnum()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('Ala ma kota'.isalnum())` → False

metoda `isalpha()`:

- sprawdza, czy łańcuch zawiera wyłącznie litery
- `str.isalpha()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('Alamakota'.isalpha())` → True

metoda `isdigit()`:

- sprawdza, czy łańcuch zawiera wyłącznie cyfry dziesiętne
- `str.isdigit()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('20160101'.isdigit()) → True`

metoda islower():

- sprawdza, czy łańcuch zawiera wyłącznie małe litery
- `str.lower()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('Alamakota'.islower())` → False

metoda `isspace()`:

- sprawdza, czy łańcuch zawiera wyłącznie tzw. białe znaki
- `str.isspace()`:
 - argument: żaden
 - wynik: True / False
- np:
`print(' \t\r\n'.isspace())` → True

metoda join(lista):

łączy elementy listy w jeden napis, rozdzielając je łańcuchem str;

elementy listy **muszą być napisami!**

- `str.join(list):`

- argument: list: pewna lista

- wynik: łańcuch powstały przez połączenie elementów listy, rozdzielonych łańcuchem str

np:

- `print(', '.join(['1', '2', '3']))` → `'1,2,3'`

funkcja list():

- tworzy listę zawierającą kolejno wszystkie znaki z napisu
- `list(str)`:
 - argument: `str` – pewien napis
 - wynik: lista o długości `len(str)`, zawierająca znaki napisu
- np:
`print(list('abc'))` → `['a', 'b', 'c']`

metoda ljust(szer):

- zwraca kopię łańcucha, wyjustowaną w lewo do szerokości szer
- `str.ljust(szer):`
 - argument: szer: docelowa szerokość łańcucha
 - wynik: kopia łańcucha wyjustowana w lewo
- np:
`print(['+'+'Ala ma kota'.ljust(20)+'']) →`
`[Ala ma kota ]`

metoda `ljust(szer,wypeł)`:

- zwraca kopię łańcucha, wyjustowaną w lewo do szerokości `szer` z wypełnieniem znakiem `wypeł`
- `str.ljust(szer)`:
 - argumenty: `szer`: docelowa szerokość łańcucha
`wypeł`: znak wypełniający (dokładnie jeden!)
 - wynik: kopia łańcucha wyjustowana w lewo
- np:
`print(['+'+'Ala ma kota'.ljust(20,'*')+'']) →`
`[Ala ma kota*****]`

metoda lower():

- zwraca kopię łańcucha ze wszystkimi literami zamienionymi na małe
- `str.lower()`:
 - argumenty: żaden
 - wynik: kopia łańcucha z literami zmienionymi na małe
- np:
`print('ŁODZIA'.lower())` → `łodzią`

metoda lstrip():

- zwraca kopię łańcucha z usuniętymi wiodącymi białymi znakami
- `str.lstrip()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po usunięciu wiodących białych znaków
- np:
`print('['+' x '.lstrip()+']') → [x ]`

metoda `lstrip(zbiór)`:

- zwraca kopię łańcucha z usuniętymi wiodącymi znakami ze zbioru `zbiór`
- `str.lstrip(zbiór)`:
 - argumenty: `zbiór`: łańcuch ze znakami do usunięcia
 - wynik: kopia łańcucha po usunięciu wiodących znaków
- np:
`print('www.zut.pl'.lstrip('w.'))` → `zut.pl`

metoda `replace(stary,nowy)`:

- zwraca kopię łańcucha z podłańcuchami `stary` zamienionymi na `nowy`
- `str.replace(stary,nowy)`:
 - argumenty: `stary`: łańcuch do zamiany
`nowy`: łańcuch zamieniający
 - wynik: kopia łańcucha po zamianie
- np:
`print('to nie to'.replace('to','ty'))` →
`ty nie ty`

metoda `replace(stary,nowy,max):`

- zwraca kopię łańcucha z co najwyżej `max` podłańcuchami `stary` zamienionymi na `nowy`
- `str.replace(stary,nowy,max):`
 - argumenty: `stary`: łańcuch do zamiany
`nowy`: łańcuch zamieniający
`max`: maksymalna liczba zamian
 - wynik: kopia łańcucha po zamianie
- np:
`print('to nie to'.replace('to','ty',1))` →
`ty nie to`

metoda rfind(podłańcuch):

- szuka podłańcucha w łańcuchu (jak find(), ale szuka od końca!)
- `str.rfind(podłańcuch):`
 - argumenty: podłańcuch: szukany łańcuch
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('to jest to'.rfind('to')) → 8`

metoda rfind(podłańcuch,start):

- szuka podłańcucha od końca łańcucha do pozycji start
- `str.rfind(podłańcuch, start)`:
 - argumenty: podłańcuch: szukany łańcuch
start: pozycja, do której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('to jest tam'.rfind('to', 3)) → -1`

metoda rfind(podłańcuch,start,meta):

- szuka podłańcucha od końca łańcucha do pozycji start
począwszy od pozycji meta
- `str.find(podłańcuch, start, meta):`
 - argumenty: podłańcuch: łańcuch
start: pozycja, do której należy szukać
koniec: pozycja, od której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('to to to'.rfind('to',3,5)) → 3`

metoda rjust(szer):

- zwraca kopię łańcucha, wyjustowaną w prawo do szerokości szer
- `str.rjust(szer):`
 - argument: szer: docelowa szerokość łańcucha
 - wynik: kopia łańcucha wyjustowana w prawo
- np:

```
print(['+'+'Ala ma kota'.rjust(20)+'']) →  
[      Ala ma kota]
```

metoda `rstrip()`:

- zwraca kopię łańcucha z usuniętymi końcowymi białymi znakami
- `str.rstrip()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po usunięciu kończących białych znaków
- np:
`print('['+' x '.rstrip()+']')` → `[ x]`

metoda `rstrip(zbiór)`:

- zwraca kopię łańcucha z usuniętymi kończącymi znakami ze zbioru `zbiór`
- `str.rstrip(zbiór)`:
 - argumenty: `zbiór`: łańcuch ze znakami do usunięcia
 - wynik: kopia łańcucha po usunięciu końcowych znaków
- np:
`print('www.zut.pl'.rstrip('l.p'))` → `www.zut`

metoda split():

- zwraca listę podłańcuchów powstałą poprzez podzielenie danego łańcucha w miejscach, w których występują białe znaki
- `str.split()`:
 - argumenty: żaden
 - wynik: tablica podłańcuchów
- np:
`print('ala ma kota'.split())` →
`['ala', 'ma', 'kota']`

metoda startswith(przedrostek):

- sprawdza, czy łańcuch zaczyna się ciągiem znaków przedrostek
- `str.startswith(przedrostek):`
 - **argumenty:** przedrostek: łańcuch
 - **wynik:** True / False
- **np:**
`print('Ala ma kota'.startswith('ta'))` → False

metoda strip():

- zwraca kopię łańcucha z usuniętymi wiodącymi i końcowymi białymi znakami
- `str.strip()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po usunięciu wiodących i kończących białych znaków
- np:
`print('['+' x '.strip()+']')` → `[x]`

metoda swapcase():

- zwraca kopię łańcucha z małymi literami zamienionymi na wielkie i odwrotnie
- `str.swapcase()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po zamianie liter małe $\Leftrightarrow$ wielkie
- np:
`print('Łódź'.swapcase())` → ŁÓDŻ

metoda title():

- zwraca kopię łańcucha z pierwszymi literami słów zamienionymi na wielkie i pozostałymi literami zamienionymi na małe
- `str.title()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po „tytularyzacji“
- np:
`print('most NA rzece KWAI'.title())` →
Most Na Rzece Kwai

metoda upper():

- zwraca kopię łańcucha ze wszystkimi literami zamienionymi na wielkie
 - `str.upper()`:
 - argumenty: żaden
 - wynik: kopia łańcucha z literami zmienionymi na wielkie
- np:
`print('łodzią'.upper())` → ŁODZIA

Porównywanie łańcuchów:

- łańcuchy można porównywać ze sobą niemal tak samo jak liczby, przy użyciu tych samych operatorów:
== != > >= < <=
- porównania typu większy-mniejszy zakładają, że do ustalenia relacji wykorzystuje się kody znaków ASCII/UNICODE, co może dawać zaskakujące rezultaty

Zapamiętaj:

- przy wykonywaniu porównań miej świadomość **typu** porównywanych danych
- w szczególności nie próbuj porównywać łańcuchów z liczbami – to może skończyć się **katastrofą**
- jedynymi dopuszczalnym porównaniami łańcucha z liczbą są `==` i `!=`
- pierwszy zawsze daje `False`, drugi zawsze `True`

Porównywanie łańcuchów i liczb:

- `'10' == 10` → `False`
- `'10' != 10` → `True`
- `'10' == 1` → `False`
- `'10' != 1` → `True`
- `'10' > 10` → **BŁĄD**

Sortowanie → funkcja sorted()

- sekwencja może być argumentem **funkcji** sorted()
- funkcja sorted() zwraca **posortowaną kopię** oryginalnej sekwencji
- sama sekwencja pozostaje **niezmieniona**

Sortowanie → metoda sort()

- każda sekwencja ma **metodę** sort()
- metoda sort() **sortuje** oryginalną sekwencję
- metoda sort() **nie zwraca** wyniku

Konwersja liczba → łańcuch

- liczbę można zamienić na reprezentujący ją łańcuch przy użyciu funkcji `str()`
- takie przekształcenie jest możliwe **zawsze**

```
str(0)           → '0'  
str(-1)          → '-1'  
str(3.14)        → '3.14'
```

Konwersja łańcuch → liczba

- łańcuch można zamienić na odpowiadającą mu liczbę używając znanych już funkcji `int()` oraz `float()`
- ...ale takie przekształcenie nie zawsze jest możliwe

Funkcje

Definicja funkcji w Pythonie:

```
def nazwa():  
    kod
```

- Wywołanie funkcji, której definicja **nie jest znana w miejscu wywołania**, kończy się **rzuceniem wyjątku**

Zapamiętaj:

- posiadanie funkcji i zmiennej o tej samej nazwie kończy się katastrofą

definicje funkcji można swobodnie przeplatać z resztą kodu:

```
print('Zaczynamy')  
  
def komunikat():  
    print('Podaj następną daną')  
  
komunikat()  
print('To koniec')
```

```
Zaczynamy  
Podaj następną daną  
To koniec
```

zapamiętaj:

- jeżeli masz zmienną, która nazywa się tak, jak pewien parametr w pewnej funkcji, to zadziała mechanizm nazywany **przesłanianiem**
- w czasie pracy tej funkcji nazwa parametru **przesłoni** nazwę zmiennej
- nazwa tej zmiennej zostanie **odsłonięta**, gdy funkcja zakończy pracę

parametryzacja pozycyjna:

- Python domyślnie posługuje się pozycyjnym przekazywaniem argumentów (i-ty argument jest kojarzony z i-tym parametrem).

nazwane przekazywanie parametrów:

- ponadto Python oferuje nazwane przekazywanie argumentów

Użycie nazwanych parametrów na prostym przykładzie:

```
def powitanie(imię,nazwisko):  
    print('Dzień dobry, nazywam się ' + \  
          nazwisko + '... ' + imię + ' ' + \  
          nazwisko + '...')  
  
powitanie(nazwisko='Bond', imię='James')  
powitanie(imię='Dolores', nazwisko='Miednica')
```

Dzień dobry, nazywam się Bond... James Bond...
Dzień dobry, nazywam się Miednica... Dolores Miednica...

Podanie nieistniejącej nazwy parametru spowoduje kłopoty...

```
def powitanie(imię,nazwisko):  
    print('Dzień dobry, nazywam się ' + \  
          nazwisko + '... ' + imię + ' ' + \  
          nazwisko + '...')  
  
powitanie(imie='Dolores', nazwisko='Miednica')
```

```
Traceback (most recent call last):  
  File "r.py", line 6, in <module>  
    powitanie(imie='Dolores', nazwisko='Miednica')  
TypeError: powitanie() got an unexpected keyword argument 'imie'
```

mieszane przekazywanie parametrów:

- możliwe jest również **mieszanie** obu stylów przekazywania parametrów
- wymaga się jednak, aby parametry **pozycyjne** pojawiły się **przed nazwanymi**

Prosta funkcja z trzema parametrami:

```
def suma(a,b,c):  
    print(    str(a) + '+' + \  
              str(b) + '+' + \  
              str(c) + '=' + str(a+b+c))
```

Wywołanie czysto pozycyjne:

```
suma(1, 2, 3)
```

$1+2+3=6$

Wywołanie czysto nazwane:

```
suma(c=1, a=2, b=3)
```

$2+3+1=6$

A teraz wywołanie mieszane:

```
suma(3, c=1, b=2)
```

Argument dla parametru **a**
jest przekazany jako
pozycyjny

Argumenty dla parametrów **b** i
c są przekazane jako
nazwane

$$3+2+1=6$$

parametry domyślne:

- wybranym parametrom można nadać **wartości domyślne** → w rezultacie można pomijać odpowiadające im argumenty, o ile ich wartość domyślna jest akceptowalna

instrukcja return – postać pierwsza:

return

- powoduje natychmiastowe **zakończenie** wykonywania funkcji i **powrót** do miejsca, z którego została wywołana
- jeśli funkcja ma **tylko efekt**, jawne użycie instrukcji return **nie jest obowiązkowe** → return zostanie ona wykonane niejawnie w miejscu, gdzie funkcja się kończy

instrukcja return – postać druga:

return *wyrażenie*

- powoduje natychmiastowe **zakończenie** wykonywania funkcji i **powrót** do miejsca, z którego została wywołana
- ponadto, funkcja **zwróci** jako swój **wynik** wartość wyrażenia stojącego za return

zapamiętaj:

- zawsze wolno **zignorować** ci wynik zwrócony przez funkcję i zadowolić się jej efektem (o ile funkcja ma jakiś efekt)

zapamiętaj:

- jeżeli pewna funkcja ma zwrócić użyteczny wynik, musi zawierać instrukcję `return` w postaci z wyrażeniem

None

- dana o wartości **None** jest kompletnie bezużyteczna
 - nie reprezentuje **żadnej** sensownej wartości
- w konsekwencji, taka dana nie może brać udziału w **żadnych** obliczeniach

Zobaczmy, jak to działa:

```
>>> print(None + 2)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#0>", line 1, in <module>
```

```
    print(None + 2)
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

dana None:

- daną None **można przypisywać** do zmiennych, aby wskazać, że zmienna nie ma użytecznej zawartości (jest „pusta”)
- zmienne można również **porównywać** z None w celu zdiagnozowania ich stanu

- **zapamiętaj:**

- jeśli pewna funkcja **nie zwróci jawnie** wartości z
użyciem

`return wyrażenie`

to funkcja taka **niejawnie zwróci wartość**

`None`

Przeanalizujmy teraz kwestię zasięgu nazw zmiennych

- **zasięgiem** nazwy zmiennej nazywamy te wszystkie **miejsca w programie**, w których **możemy korzystać** z tej zmiennej
- wiemy już, że zasięgiem **parametru funkcji** jest ta **funkcja** (i nic poza nią)

A jaki jest zasięg zmiennej używanej poza funkcją?

- przeprowadzimy kilka eksperymentów, które pokażą nam jak Python konstruuje zasięgi zmiennych
- pierwszy z nich sprawdzi, czy zmienna **zdefiniowana poza funkcją** jest widoczna w funkcji

Przeanalizujmy taki kod:

```
def funkcja():  
    print("Czy ja znam tę zmienną? ", zmienna)  
  
zmienna = 1  
funkcja()  
print(zmienna)
```

Jest OK!

```
Czy ja znam tę zmienną? 1  
1
```


**A co się stanie, jeśli funkcja zdefiniuje
zmienną o takiej samej nazwie?**

Wprowadźmy drobną modyfikację:

```
def funkcja():  
    zmienna=2 ←  
    print("Czy ja znam tę zmienną? ", zmienna)
```

```
zmienna = 1  
funkcja()  
print(zmienna)
```

Jest inaczej,
ale OK!

```
Czy ja znam tę zmienną? 2 ←  
1
```

Skupiamy uwagę, bo teraz stanie się coś ważnego:

- zrobimy poprawkę tak drobną, że prawie niezauważalną!

Wprowadźmy drobną modyfikację:

```
def funkcja():  
→ print("Czy ja znam tę zmienną? ", zmienna)  
→ zmienna=2
```

```
zmienna = 1  
funkcja()  
print(zmienna)
```

Co się stało
się?

```
Traceback (most recent call last):  
  File "r.py", line 6, in <module>  
    funkcja()  
  File "r.py", line 2, in funkcja  
    print("Czy ja znam tę zmienną? ", zmienna)  
UnboundLocalError: local variable 'zmienna' referenced before assignment
```

zapamiętaj:

- jeżeli funkcja używa nazwy pewnej zmiennej, to nazwa ta **przesłania** zasięg zmiennej zdefiniowanej na **zewnątrz tej funkcji**
- przesłonięcie działa w **całej funkcji**, nawet zanim zmiennej takiej nada się wartość po raz pierwszy

zapamiętaj:

- będąc w funkcji, nie jesteśmy w stanie (poznanymi dotąd środkami) zmienić wartości zmiennej zdefiniowanej poza funkcją

instrukcja global:

`global nazwa`

`global nazwa2, nazwa2, ...`

- informuje Pythona, że nie życzymy sobie przesłonięcia nazwy zewnętrznej

zapamiętaj:

- jeśli funkcja oznaczy pewną zmienną jako **globalną**, to tym **zrezygnuje z przestłaniania** tej zmiennej
- zasięg tej zmiennej obejmie więc **całą funkcję**

Wyjątki

wyjątek (ang. exception):

- **wyjątek** to specyficzna dana, powstająca w sposób *automagiczny* w chwili **wystąpienia błędu**, który uniemożliwia kontynuowanie wykonania programu
- w Pythonie nazywa się to **podniesieniem** (ang. raise) wyjątku

Prosty przykład:

```
a = int(input())  
b = int(input())  
try:  
    print(a / b)  
except:  
    print('Nic z tego!')  
print('To koniec')
```

Potem sprawdzamy:

```
a = int(input())
b = int(input())
try:
    print(a / b)
except:
    print('Nic z tego!')
print('To koniec')
```

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

Selektywna obsługa wyjątków:

```
try:
    x = int(input())
    y = 1 / x
→ except ZeroDivisionError:
    print('nie potrafię dzielić przez zero!')
→ except ValueError:
    print('a miałeś podać liczbę!')
→ except:
    print('0ch....')
print('To koniec')
```

zapamiętaj:

- gałęzie **except** są przeszukiwane w takiej **kolejności**, w jakiej zostały **zapisane w programie**
- liczba różnych gałęzi **except** jest **dowolna**, ale jeśli wystąpiło **try**, to musi pojawić się **choć jedno except**
- **except** nie może wystąpić bez poprzedzającego **try**

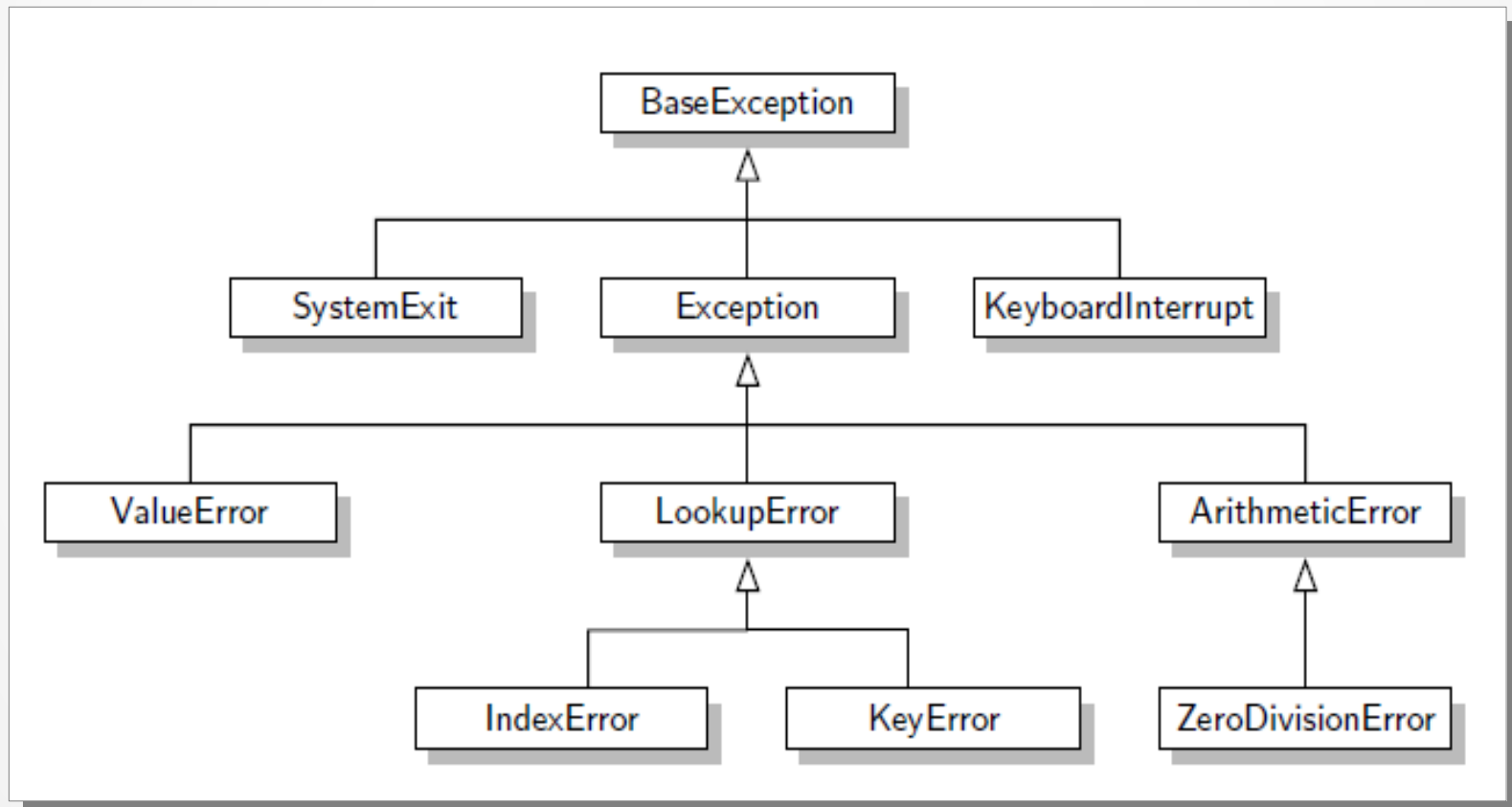
zapamiętaj:

- jeśli **wykonała się** jakaś gałąź **except**, to już nie wykona się **żadna inna**
- jeśli żadna z gałęzi **except** **nie będzie pasować** do podniesionego wyjątku, to wyjątek zostanie uznany za **nieobsłużony**
- **except bez nazwy wyjątku** (tzw. **except domyślny**) musi być wymieniony jako **ostatni**

wyjątki w Pythonie 3:

- Python 3 wyróżnia **63** wyjątki wbudowane
- wyjątki w Pythonie tworzą **hierarchię drzewiastą**
- innymi słowami, pewne są **abstrakcyjne**, a pewne **konkretne**

- Oto (skromny) fragment drzewa wyjątków Pythona:



Jeszcze jedna forma:

```
try:  
    :  
except ( exc1, exc2 ):  
    :
```

wyjątki i funkcje:

- jeśli pewien wyjątek zostanie podniesiony w funkcji, to może zostać **obsłużony**:
 - w **tej** funkcji
 - **poza** tą funkcją
 - **i tu, i tu**

Obsługa wewnątrz funkcji:

```
def badfun(n):  
    try:  
        return 1/n  
    except ArithmeticError:  
        print('Sorry, taką mamy arytmetykę!')  
        return None  
  
nn = 0  
badfun(nn)  
print('To koniec')
```

```
Sorry, taką mamy arytmetykę!  
To koniec
```

Obsługa poza funkcją:

```
def badfun(n):  
    return 1/n  
  
nn = 0  
try:  
    badfun(nn)  
except ArithmeticError:  
    print('Funkcjo, cóżeś mi uczyniła?')  
print('To koniec')
```

Funkcjo, cóżeś mi uczyniła?
To koniec

instrukcja raise:

- `raise` *wyjątek*
- powoduje podniesienie wyspecyfikowanego wyjątku tak, jakby wystąpił naprawdę

Wyjątek podniesiony na żądanie:

```
def badfun(n):  
    raise ZeroDivisionError  
  
nn = 0  
try:  
    badfun(nn)  
except ArithmeticError:  
    print('Funkcjo, cóżeś mi uczyniła?')  
print('To koniec')
```

Funkcjo, cóżeś mi uczyniła?
To koniec

instrukcja raise:

- `raise`
- powoduje podniesienie **takiego samego** wyjątku, jaki jest właśnie **obsługiwany**
- co oznacza, że instrukcji tej można użyć tylko **w ramach obsługi wyjątku**

jak działa asercja?

- instrukcja `assert` oblicza swoje wyrażenie
- jeśli jego wartością jest `True` albo wartość **różna od zera i od `None`** albo **niepusty napis**, to nie dzieje się **NIC**
- w przeciwnym przypadku zostaje **podniesiony** wyjątek `AssertionError` (mówimy wtedy, że asercja zawiodła)

wybrane wyjątki wbudowane Pythona 3

ArithmeticError

- `BaseException` ← `Exception` ← `ArithmeticError`
- wyjątek abstrakcyjny, zawierający w sobie wyjatki wywoływane przez błędy operacji arytmetycznych

AssertionError

- `BaseException` ← `Exception` ← `AssertionError`
- podnoszony przez instrukcję `assert`, gdy asercja zawodzi

BaseException

- BaseException
- wyjątek najbardziej abstrakcyjny, obejmuje sobą wszelkie możliwe wyjątki

Exception

- `BaseException` ← `Exception`
- wyjątek abstrakcyjny, zawierający w sobie te wyjątki wbudowane, które są podnoszone w wyniku błędów

FloatingPointError

•BaseException ← Exception ← ArithmeticError ← FloatingPointError

- podnoszony przez błędy operacji arytmetycznych na liczbach rzeczywistych

IndexError

- `BaseException` ← `Exception` ← `LookupError` ← `IndexError`
- podnoszony przez użycie niepoprawnego indeksu sekwencji

KeyboardInterrupt

- `BaseException` ← `KeyboardInterrupt`
- podnoszony w wyniku użycia kombinacji klawiszy przerywających program (najczęściej Ctrl-C)

LookupError

- `BaseException` ← `Exception` ← `LookupError`
- wyjątek abstrakcyjny, zawierający w sobie wszystkie wyjątki wywoływane przez błędy dostępu do tzw. kolekcji (sekwencje są jednymi z kolekcji)

MemoryError

- `BaseException` ← `Exception` ← `MemoryError`
- podnoszony, gdy operacja nie może zostać zakończona z powodu braku dostępnej pamięci

OverflowError

- `BaseException` ← `Exception` ← `ArithmeticError` ← `OverflowError`
- podnoszony, gdy wynik operacji arytmetycznej jest zbyt duży (co do modułu), by można go było zachować

RuntimeError

- `BaseException` ← `Exception` ← `RuntimeError`
- wyjątek abstrakcyjny, zawierający w sobie te wyjątki wbudowane, które są podnoszone w na skutek błędów w wykonaniu kodu

RecursionError

- `BaseException` ← `Exception` ← `RuntimeError` ← `RecursionError`
- podnoszona na skutek (potencjalnie) nieskończonej rekurencji

sekwencja (*sequence*)

- rodzaj danej w Pythonie
- uporządkowana kolekcja wartości
- sekwencjami są listy i napisy
- dostęp do odczytu elementów sekwencji zapewniają:
 - indeksowanie
 - wycinki
 - pętla for

mutowalność (*mutability*)

- własność danej w Pythonie
- dana mutowalna (*mutable*) może być swobodnie modyfikowania w czasie swojego życia
- dana niemutowalna (*immutable*) nie zmienia swojej wartości w czasie swojego życia
- mutowalne są **listy**
- niemutowalne są **napisy**

iterowalność (*iterability*)

- własność kolekcji danych w Pythonie
- dana iterowalna może pojawić się po prawej stronie operatora `in` w pętli `for`
- iterowalne są **listy** i **napisy**

krotka (*tuple*)

- krotka jest **niemutowalną sekwencją**
- w niektórych aspektach swojego zachowania przypomina listę (ale z uwzględnieniem niemutowalności)

krotka

- jednym ze sposobów utworzenia krotki jest podanie wszystkich jej wartości rozdzielonych przecinkami i ujętych w nawiasy okrągłe

tworzenie krotek

```
tup1 = (1,2,3)
tup2 = ('ala', 'ma', 'kota')
tup3 = (1, 2.3, 'krotka')
print(tup1)
print(tup2)
print(tup3)
```

```
(1, 2, 3)
('ala', 'ma', 'kota')
(1, 2.3, 'krotka')
```

krotka

- jeżeli krotka składa się wyłącznie z jednej wartości, to przecinka następującego po niej nie wolno pominąć (dlaczego?)

tworzenie krotek

```
tup1 = (1,)  
tup2 = ('ala',)  
print(tup1)  
print(tup2)
```

```
(1,)  
('ala',)
```

krotka

- jeżeli krotka ma być argumentem funkcji, otaczających ją nawiasów nie wolno pominąć (dlaczego?)

tworzenie krotek

```
def f(t):  
    return t[0]+t[1]  
  
print(f((1,2,3)))
```

```
(1,)  
( 'ala', )
```


krotka

- choć brzmi to nieco szokująco, nawiasy okrągłe można pominąć, jednak może to źle wpłynąć na czytelność kodu

tworzenie krotek

```
tup1 = 1,2,4  
tup2 = 'ala',  
print(tup1)  
print(tup2)
```

```
(1,2,4)  
( 'ala', )
```

krotka

- stworzenie pustej krotki **zawsze** wymaga nawiasów

tworzenie krotek

```
tup1 = ()  
print(tup1)
```

()

krotka

- krotka może zawierać w sobie inne krotki

krotki w krotkach

```
tup1 = (1, (1,2), (3,4,5))  
print(tup1)
```

```
(1, (1, 2), (3, 4, 5))
```

krotka

- krotka może zawierać listy, a listy mogą zawierać krotki

krotki w krotkach

```
tup1 = (1, [2,3])  
lst1 = [1, (2,3)]  
print(tup1)  
print(lst2)
```

```
(1, [2, 3])  
[1, (2, 3)]
```


krotka

- w aspekcie odczytu zawartych w niej elementów krotka zachowuje się jak lista

liczba elementów w krotce

```
tup0 = ()  
tup1 = (1,)  
tup2 = (2,3)  
print(len(tup0))  
print(len(tup1))  
print(len(tup2))
```

```
0  
1  
2
```

indeksowanie krotki

```
tuple = (1, 10, 100, 1000)
print(tuple[0])
print(tuple[-1])
```

1

1000

wycinki krotki

```
tuple = (1, 10, 100, 1000)
print(tuple[1:])
print(tuple[:-2])
```

```
(10, 100, 1000)
(1, 10)
```

iterowanie po krotce

```
tuple = (1, 10, 100, 1000)
for el in tuple:
    print(el)
```

```
1
10
100
1000
```

krotka

- ponieważ krotka jest niemutowalna, analogie w do zachowania list w tym momencie się kończą

wszystkie poniższe próby modyfikowania krotki są nielegalne!

```
tuple = (1, 10, 100, 1000)  
tuple.append(10000)  
del tuple[0]  
tuple[1] = -10
```

krotka

- operator `+` potrafi konkatelować krotki, dając w wyniku nową krotkę

konkatenowanie krotek

```
tuple1 = (1, 10, 100, 1000)  
tuple2 = (0,)  
print(tuple1 + tuple2)
```

(1, 10, 100, 1000, 0)

krotka

- niemutowalność krotek **nie oznacza**, że nie wolno im nadać nowej wartości
- oznacza tylko, że nie wolno modyfikować ich bieżącej wartości

konkatenowanie krotek

```
tuple1 = (1, 10, 100, 1000)
tuple2 = (0,)
tuple1 = tuple1 + tuple2
print(tuple1)
```

(1, 10, 100, 1000, 0)

krotka

- operator $*$ potrafi multiplikować krotkę, dając w wyniku nową krotkę

multiplikowanie krotek

```
tuple = (1, 10, 100)  
t2 = tuple * 3  
print(t2)
```

```
(1, 10, 100, 1, 10, 100, 1, 10, 100)
```

krotka

- operatory `in` oraz `not in` działają również w odniesieniu do krotek

multiplikowanie krotek

```
tuple = (1, 10, 100)
print(10 in tuple)
print(-10 not in tuple)
```

True

True

krotka

- krotki mogą się pojawiać również po lewej stronie instrukcji podstawienia
- uwaga: nie narusza to w żaden sposób faktu ich niemutowalności

krotki po lewej stronie operatora podstawienia

```
v1 = 1  
v2 = 2  
v3 = 3  
v1, v2, v3 = v2, v3, v1  
print(v1, v2, v3)
```

2 3 1

krotka

- krotkę można przekształcić w listę, używając funkcji `list()`
- listę można przekształcić w krotkę, używając funkcji `tuple()`

konwersje krotka ↔ lista

```
lst = [1,2,3]
tup = 4,5,6
lst2 = list(tup)
tup2 = tuple(lst)
print(lst2)
print(tup2)
```

```
[4, 5, 6]
(1, 2, 3)
```

słownik (*dictionary*)

- słownik jest daną **mutowalną**
- słownik nie jest **sekwencją**
- słownik jest kolekcją par **klucz:wartość**
- słownik jest dość wiernym odbiciem perlowej koncepcji *hasza*

tworzenie słownika

```
dct = {'cat': 'chat', 'dog': 'chien', 'horse': 'cheval'}  
phones = { 'boss' : 5551234567, 'Suzy' : 5557654321 }  
empty = { }
```

```
print(dct)  
print(phones)  
print(empty)
```

```
{ 'dog': 'chien', 'horse': 'cheval', 'cat':  
  'chat' }  
{ 'Suzy': 5557654321, 'boss': 5551234567 }  
{ }
```

słownik

- słownik można również utworzyć z dowolnej sekwencji, zawierającej pary wartości – czyni się to przy użyciu funkcji `dict()`

tworzenie słownika z listy i z krotki

```
dct = dict([('cat', 'chat'), ('dog', 'chien')])  
print(dct)  
dct = dict (('cat', 'chat'), ('dog', 'chien'))  
print(dct)
```

```
{'dog': 'chien', 'cat': 'chat'}  
{'dog': 'chien', 'cat': 'chat'}
```

słownik

- możliwe jest również użycie składni „kluczowej”, ale tylko wtedy, gdy klucze są łańcuchami nie zawierającymi znaków nie-alfanumerycznych (wyglądają jak nazwy zmiennych)
- uproszczenie zapisu dotyczy kluczy, ale **nie** wartości

tworzenie słownika w stylu „kluczowym”

```
dct = dict(cat='chat', dog='chien')  
print(dct)
```

```
{'dog': 'chien', 'cat': 'chat'}
```

słownik

- dostęp do wartości słownika składniowo wygląda jak indeksowanie listy

dostęp po kluczu

```
dct = {'cat': 'chat', 'dog': 'chien', 'horse': 'cheval'}  
phones = { 'boss' : 5551234567, 'Suzy' : 5557654321 }  
print(dct['cat'])  
print(phones['Suzy'])
```

chat

5557654321

słownik

- **uwaga:** próba dostępu do nieistniejącej wartości klucza kończy się podniesieniem wyjątku `KeyError`
- na szczęście, operator `in` pozwala efektywnie sprawdzić obecność klucza w słowniku

sprawdzanie obecności klucza

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
words = [ 'cat', 'lion', 'horse' ]  
for word in words:  
    if word in dct:  
        print(word, "->", dct[word])  
    else:  
        print("Not in dictionary: ", word)
```

```
cat -> chat  
Not in dictionary:  lion  
horse -> cheval
```

słownik

- ponieważ słowniki nie są sekwencjami, nie można iterować po nich pętlą `for`
- można jednak iterować po kluczach istniejących w słowniku dzięki istnieniu metody `keys()` (skojarz z funkcją `keys()` z Perla)
- metoda `keys()` zwraca listę kluczy ułożonych w nieprzewidywalnej kolejności

iterowanie po kluczach

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
for key in dct.keys():  
    print(key, "->", dct[key])  
for key in sorted(dct.keys()):  
    print(key, "->", dct[key])
```

```
horse -> cheval  
dog -> chien  
cat -> chat  
cat -> chat  
dog -> chien  
horse -> cheval
```

słownik

- można również iterować po parach (klucz, wartość), co umożliwia metoda `items()`
- metoda `items()` zwraca listę dwuelementowych krotek

iterowanie po krotkach (klucz,wartość)

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
for en, fr in dct.items():  
    print(en, "->", fr)
```

```
cat -> chat  
dog -> chien  
horse -> cheval
```

słownik

- możliwe jest także iterowanie po wartościach, co umożliwia metoda `values()`

iterowanie po wartościach

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
for fr in dct.values():  
    print(fr)
```

```
cheval  
chien  
chat
```

słownik

- fakt, że słowniki są mutowalne, otwiera drogę do ich swobodnego modyfikowania w trakcie ich życia

zmiana wartości istniejącego klucza

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
dct['cat'] = 'minou'  
print(dct)
```

```
{'dog': 'chien', 'horse': 'cheval', 'cat':  
'minou'}
```

dodanie nowego klucza

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
dct['lion'] = 'lion'  
print(dct)
```

```
{'lion': 'lion', 'horse': 'cheval', 'dog': 'chien', 'cat': 'chat'}
```

usuwanie pary klucz:wartość

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
del dct['dog']  
print(dct)
```

```
{'cat': 'chat', 'horse': 'cheval'}
```

słownik

- **uwaga:** próba usunięcia nieistniejącego klucza powoduje podniesienie wyjątku
KeyError

słownik

- chociaż słowniki nie są sekwencjami, to funkcja `len()` akceptuje je jako argument i zwraca liczbę zgromadzonych w słowniku par klucz:wartość

„długość” słownika

```
dct = { 'cat': 'chat', 'dog': 'chien', 'horse': 'cheval' }  
print(len(dct))
```

3

zbiór (*set*)

- zbiór w Pythonie jest praktyczną realizacją zbioru w sensie matematycznym, tzn. jest uporządkowaną kolekcją **unikalnych** wartości (każdy element może wystąpić w zbiorze co najwyżej raz)
- zbiór jest **mutowalny**, ale nie jest **sekwencją**, aczkolwiek jest **iterowalny**

zbiór

- uwaga: elementy w zbiorze układane w są w nieprzewidywalnej kolejności

tworzenie zbioru

```
set1 = { 3, 8, 1, 5 }  
set2 = { 'gamma', 'beta', 'alfa' }  
print(set1)  
print(set2)
```

```
{8, 1, 3, 5}  
{'beta', 'gamma', 'alfa'}
```

zbiór

- uwaga: zbiór pusty tworzy się wywołaniem funkcji `set()` (bez argumentów)! Użycie formy `{}` utworzy pusty słownik!

tworzenie zbioru

```
empty = set()  
print(empty)
```

set()

zbiór

- zbiór można utworzyć z dowolnej sekwencji, również poprzez użycie funkcji `set()`
- ewentualnie istniejące powtarzające się elementy zostaną usunięte

tworzenie zbioru

```
set1 = set([1,2,3,1])  
set2 = set("alamakota")  
print(set1)  
print(set2)
```

{1, 2, 3}

{'a', 'o', 't', 'l', 'm', 'k'}

zbiór

- zbiór można utworzyć również ze słownika, ale w takim przypadku tworzy się wyłącznie zbiór kluczy (oczywiście, można utworzyć zbiór z wartości, ale trzeba jawnie użyć metody `values()`)

tworzenie zbioru

```
dct = { 1:2, 3:4, 5:6 }  
set = set(dct)  
print(set)
```

{1, 3, 5}

zbiór

- operatory bitowe znane z „C” (&, |, ^) w przypadku używania ze zbiorami stają się operatorami zbiorowymi (wraz z –)

operacje na zbiorach

```
set1 = {1,2,3}  
set2 = {2,3,4}  
set = set1 | set2  
print(set)  
set = set1 & set2  
print(set)  
set = set1 ^ set2  
print(set)  
set = set1 - set2  
print(set)
```

```
{1, 2, 3, 4}  
{2, 3}  
{1, 4}  
{1}
```

zbiór

- zbiory są mutowalne, co pozwala zmieniać ich zawartość w trakcie ich życia

operacje na zbiorach

```
set = {0}  
set |= {1}  
print(set)  
set &= {0}  
print(set)  
set ^= {1}  
print(set)  
set -= set  
print(set)
```

```
{0, 1}  
{0}  
{0, 1}  
set()
```

zbiór

- pętla `for` potrafi iterować po zawartości zbioru

operacje na zbiorach

```
set = set('abc') | {'d'}  
for x in set:  
    print(x)
```

d
c
a
b

rozpoznawanie typu danej

- Python, jako język o dynamicznej typizacji, udostępnia środki pozwalające rozpoznać typ danej przechowywanej w zmiennej
- służy do tego funkcja o nazwie `type()`
- funkcja ta zwraca **obiekt** klasy **type**, opisujący typ danej

rozpoznawanie typu danej

```
a = 1
b = 2.
c = 'Mary'
d = dict( a='b')
print(type(a))
print(type(b))
print(type(c))
print(type(d))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'dict'>
```

predefiniowane typy danych (wybrane)

NoneType

bool

int

long

float

str

tuple

list

dict

rozpoznawanie typu danej

- operator **is** (stwierdzający przynależność obiektu do klasy) może posłużyć do stwierdzenie zgodności typu zmiennej z typem pożądanym

rozpoznawanie typu danej

```
a = 1
b = 2.
c = 'Mary'
d = dict( a='b' )
print(type(a) is float)
print(type(b) is float)
print(type(c) is list)
print(type(d) is dict)
```

```
False
True
False
True
```

rozpoznawanie typu danej

- obiekt klasy `type` ma własność o nazwie `__name__`, która przechowuje nazwę typu jako łańcuch

rozpoznawanie typu danej

```
a = 1
b = 2.
c = 'Mary'
d = dict( a='b' )
print(type(a).__name__)
print(type(b).__name__)
print(type(c).__name__)
print(type(d).__name__)
```

```
int
float
str
dict
```


funkcje anonimowe

- Python umożliwia tworzenie funkcji anonimowych, które mogą być tworzone *ad hoc* w miejscach, w których są potrzebne
- używa się do tego celu słowa kluczowego `lambda`

funkcje anonimowe

```
def f(x):  
    return x*x  
  
g = lambda x: x*x  
print(f(2))  
print(g(2))
```

4

4

funkcje anonimowe

- „ciało” funkcji anonimowej nie zawiera instrukcji **return**
- wartością takiej funkcji jest **wyłącznie** wartość zawartego w niej wyrażenia
- oznacza to, że funkcja taka **zawsze** zwraca wartość
- funkcja taka może mieć dowolną liczbę parametrów

funkcje anonimowe

```
g = lambda x,y: x**y  
print(g(2,3))
```

8

funkcje anonimowe

- pełna użyteczność funkcji anonimowych ujawnia się we współdziałaniu z innymi funkcjami
- np. w przypadku sortowania funkcją `sorted()`

funkcje anonimowe

```
people = [('John', 23), ('Adam', 16), ('Zed', 82)]  
print(sorted(people))
```

```
[('Adam', 16), ('John', 23), ('Zed', 82)]
```

funkcje anonimowe

```
people = [('John', 23), ('Adam', 16), ('Zed', 82)]  
print(sorted(people, key=lambda who:who[-1]))
```

```
[('Adam', 16), ('John', 23), ('Zed', 82)]
```

funkcje anonimowe

- funkcja `map()`: wykonuje pewną czynność na wszystkich elementach sekwencji, zwracając jak wynik nową sekwencję

funkcje anonimowe

```
people = [('John', 23), ('Adam', 16), ('Zed', 82)]  
upperpeople = list(map(lambda x: (x[0].upper(),x[1]), people))  
print(people)  
print(upperpeople)
```

```
[('John', 23), ('Adam', 16), ('Zed', 82)]  
[('JOHN', 23), ('ADAM', 16), ('ZED', 82)]
```

„list comprehension“

- konwencja składniowa umożliwiająca zwarte i czytelne kreowanie sekwencji (głównie list, ale nie tylko)
- np. trywialne zadanie stworzenia listy wypełnionej pięcioma kolejnymi liczbami naturalnymi

„list comprehension“

```
list = []  
for i in range(5):  
    list.append(i+1)  
print(list)
```

[1, 2, 3, 4, 5]

„list comprehension“

```
list = [ i+1 for i in range(5) ]  
print(list)
```

[1, 2, 3, 4, 5]

„list comprehension“

```
L1 = [x**2 for x in range(10)]  
L2 = [x for x in L1 if x % 2 == 0]  
print(L1)  
print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
[0, 4, 16, 36, 64]
```

„list comprehension“

```
noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]  
primes = [x for x in range(2, 50) if x not in noprimes]  
print(primes)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Programowanie obiektowe w Pythonie

Najprostsza możliwa klasa:

```
class OurClass:  
    pass
```

Programowanie obiektowe w Pythonie

Utworzenie obiektu klasy:

```
ourob = OurClass()
```


Konstruktor klasy:

- jeśli klasa ma mieć konstruktor, musi on przybrać postać funkcji o nazwie `__init__`
- konstruktor wołany jest niejawnie w chwili tworzenia obiektu klasy
- każda funkcja klasy (metoda) musi mieć co najmniej jeden parametr – pierwszy z nich jest zawsze niejawnie i automatycznie przekazaną referencją do obiektu
- zwyczajowo parametr ten nazywa się `self`

Programowanie obiektowe w Pythonie

Utworzenie obiektu klasy:

```
class Stack:  
    def __init__(self):  
        print('Hi')  
  
stack = Stack()
```

Hi!

Konstruktor klasy:

- jeśli konstruktor utworzy (używając self) własności wewnątrz obiektu, będą one istnieć w każdym obiekcie
- domyślnie własności obiektu są publiczne

Programowanie obiektowe w Pythonie

Utworzenie własności w obiekcie klasy:

```
class Stack:
    def __init__(self):
        self.stk = []

stack = Stack()
print(len(stack.stk))
```

Właściwości klasy:

- jeśli pewna własność ma być prywatna, jej nazwa musi rozpoczynać się od __
- prywatność takiej metody jest jednak umowna – dostęp do niej i tak jest możliwy (choć trochę utrudniony)

Programowanie obiektowe w Pythonie

Utworzenie własności *prywatnej* w obiekcie klasy:

```
class Stack:  
    def __init__(self):  
        self.__stk = []  
  
stack = Stack()  
print(len(stack.__stk))
```

```
AttributeError: 'Stack' object has no attribute  
'__stk'
```

Metody klasy:

- domyślnie wszystkie metody klasy są publiczne
- wszystkie metody traktują swój pierwszy parametr jako obiekt, na rzecz którego są wywołane (self)
- przy wywołaniu nie wolno podawać argumentu dla tego parametru

Programowanie obiektowe w Pythonie

Utworzenie metod w obiekcie klasy:

```
class Stack:
    def __init__(self):
        self.__stk = []

    def push(self, val):
        self.__stk.append(val)

    def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val
```


Programowanie obiektowe w Pythonie

Wywołanie metod z obiektu klasy:

```
stack = Stack()  
stack.push(3)  
stack.push(2)  
stack.push(1)  
print(stack.pop())  
print(stack.pop())  
print(stack.pop())
```

1
2
3

Dziedziczenie:

- Python wspiera dziedziczenie wielobazowe, jednak czyni to w sposób bardzo odbiegający od tego, co znamy np. z C++
- zdefiniowanie podklasy wymaga podania nazwy nadklasy (nadklas)

Programowanie obiektowe w Pythonie

Utworzenie podklasy:

```
class AddingStack(Stack):  
    pass
```

Dziedziczenie:

- podklasa dziedziczy cały „dobytek” nadklasy, zachowując dostęp ustalony w nadklasie
- ciekawym zjawiskiem jest natomiast fakt, że nigdy nie ma miejsca domyślne wywołanie konstruktora nadklasy – trzeba go wywołać jawnie
- co więcej, w tym przypadku trzeba przekazać do konstruktora self
- wywołanie konstruktora z nadklasy wymaga podania jej nazwy

Programowanie obiektowe w Pythonie

Przedefiniowanie metod

```
class AddingStack(Stack):  
    def __init__(self):  
        Stack.__init__(self)  
        self.__sum = 0
```

Dziedziczenie:

- podklasa zachowuje pełnię praw dostępu do prywatnych komponentów nadklasy
- zdefiniowanie metody o nazwie znanej w nadklasie powoduje jej przesłonięcie
- dostęp do oryginału wymaga podania nazwy nadklasy oraz jawnego przekazania argumentu self

Programowanie obiektowe w Pythonie

Przedefiniowanie metod

```
class AddingStack(Stack):  
    def __init__(self):  
        Stack.__init__(self)  
        self.__sum = 0  
    def push(self, val):  
        self.__sum += val  
        Stack.push(self, val)  
    def pop(self):  
        val = Stack.pop(self)  
        self.__sum -= val  
        return val
```

Dziedziczenie:

- dostęp do prywatnych komponentów z zewnątrz klasy można realizować przy pomocy „getterów” i „setterów”

Programowanie obiektowe w Pythonie

Przedefiniowanie metod

```
class AddingStack(Stack):  
    def __init__(self):  
        Stack.__init__(self)  
        self.__sum = 0  
    def push(self, val):  
        self.__sum += val  
        Stack.push(self, val)  
    def pop(self):  
        val = Stack.pop(self)  
        self.__sum -= val  
        return val  
    def getSum(self):  
        return self.__sum
```

Własności :

- własności w obiektach Pythona dzielą się na dwa rodzaje:
- **instancyjne** (*instance variables*) – przechowywane w obiekcie
- **klasowe** (*class variables*) – przechowywane poza obiektami klasy w jednym egzemplarzu

Własności :

- uwaga – każdy obiekt danej klasy może mieć inny zestaw własności instancyjnych (!)
- każdy obiekt ma niejawnie dziedziczoną własność o nazwie `__dict__` zawierającą nazwy własności danego obiektu i ich wartości
- wynika stąd pośrednio wniosek, że obiekt w Pythonie jest słownikiem
- własności instancyjne można tworzyć w dowolnym momencie (nie tylko w konstruktorze!)

Programowanie obiektowe w Pythonie

Własności instancyjne

```
class Class:
    def __init__(self, val=1):
        self.First = val
    def setSecond(self, val):
        self.Second = val
```

```
object1 = Class()
object2 = Class(2)
object2.setSecond(3)
object3 = Class(4)
object3.Third = 5
```

```
print(object1.__dict__)
print(object2.__dict__)
print(object3.__dict__)
```

```
{'First': 1}
{'Second': 3, 'First': 2}
{'Third': 5, 'First': 4}
```

Własności:

- zbadajmy teraz, jak Python traktuje własności zdefiniowane jako prywatne (te, których nazwy zaczynają się od __)

Programowanie obiektowe w Pythonie

Własności instancyjne prywatne

```
class Class:
    def __init__(self, val=1):
        self.__First = val
    def setSecond(self, val=2):
        self.__Second = val
```

```
object1 = Class()
object2 = Class(2)
object2.setSecond(3)
object3 = Class(4)
object3.__Third = 5
```

```
print(object1.__dict__)
print(object2.__dict__)
print(object3.__dict__)
```

```
{'_Class__First': 1}
{'_Class__First': 2, '_Class__Second': 3}
{'_Class__First': 4, '__Third': 5}
```

Własności:

- widać stąd, że jeśli klasa X ma prywatną własność __Y, to dostęp do niej jest możliwy poprzez sztucznie skonstruowaną („zmanglowaną”) nazwę:

`_X__Y`

- widać z tego, że pythonowa prywatność jest bardzo **umowna**

Własności :

- własności klasowe tworzy się wewnątrz definicji klasy, ale poza definicją dowolnej z jej metod
- własności taką de facto tworzy przypisanie do niej wartości
- dostęp do takiej własności podlega takim samym zasadom, jakie dotyczą własności instancyjnych
- własności klasowe nie są uwidaczniane w zawartości własności `__dict__` obiektu

Programowanie obiektowe w Pythonie

Własności klasowe

```
class Class:
    Counter = 0
    def __init__(self, val=1):
        self.__First = val
        Class.Counter += 1

object1 = Class()
object2 = Class(2)
object3 = Class(4)

print(object1.__dict__, object1.Counter)
print(object2.__dict__, object2.Counter)
print(object3.__dict__, object3.Counter)
```

```
{'_Class__First': 1}
{'_Class__First': 2, '_Class__Second': 3}
{'_Class__First': 4, '__Third': 5}
```

Własności:

- uprywatnienie własności klasowej daje takie same skutki, jakie znamy z własności instancyjnych

Programowanie obiektowe w Pythonie

Własności klasowe

```
class Class:
    __Counter = 0
    def __init__(self, val=1):
        self.__First = val
        Class.__Counter += 1

object1 = Class()
object2 = Class(2)
object3 = Class(4)

print(object1.__dict__, object1._Class__Counter)
print(object2.__dict__, object2._Class__Counter)
print(object3.__dict__, object3._Class__Counter)
```

```
{'_Class__First': 1} 3
{'_Class__First': 2} 3
{'_Class__First': 4} 3
```

Własności:

- ponieważ przynależność obiektu X do klasy Y nie determinuje w żaden sposób nazw posiadanych przez niego własności, przy dostępie do „rzadkich” własności może być konieczna dodatkowa asekuracja

Programowanie obiektowe w Pythonie

Dostęp do własności

```
class Class:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1

object = Class(1)
print(object.a)
try:
    print(object.b)
except AttributeError:
    pass
```

Własności :

- na szczęście, istnieje łagodniejsza forma sprawdzenia dostępności własności, udostępniana przez funkcję

`hasattr(object, 'property')`

Programowanie obiektowe w Pythonie

Dostęp do własności

```
class Class:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1

object = Class(1)
print(object.a)
if hasattr(object, 'b'):
    print(object.b)
```

Własności:

- funkcja `hasattr(object, 'property')` operuje też na własnościach klasowych

Programowanie obiektowe w Pythonie

Dostęp do własności

```
class Class:  
    Attr = 1  
  
print(hasattr(Class, 'Attr'))  
print(hasattr(Class, 'Prop'))
```

True

False

Własności:

- wyniki zwracane przez tę funkcję mogą być momentami dość zaskakujące

Programowanie obiektowe w Pythonie

Dostęp do własności

```
class Class:
    a = 1
    def __init__(self):
        self.b = 2

object = Class()
print(hasattr(object, 'b'))
print(hasattr(object, 'a'))
print(hasattr(Class, 'b'))
print(hasattr(Class, 'a'))
```

True

True

False

True

Metody:

- jak już wspomnieliśmy, wywołanie metody na zewnątrz klasy, powoduje niejawne przekazanie jako pierwszego argumentu obiektu, na rzecz którego obiekt wywołano – jest to całkowicie automatyczne
- dlatego też metoda ma o jeden parametr więcej, niż podaje się argumentów w jej wywołaniu

Programowanie obiektowe w Pythonie

Metody

```
class Classy:  
    def method(self):  
        print("method")  
  
obj = Classy()  
obj.method()
```

method

Programowanie obiektowe w Pythonie

Metody

```
class Classy:  
    def method(self, par):  
        print("method:", par)  
  
obj = Classy()  
obj.method(1)
```

method

Metody:

- dostęp do własności obiektu i klasy uzyskuje się poprzez self

Programowanie obiektowe w Pythonie

Metody

```
class Classy:  
    Variable = 2  
    def method(self):  
        print(self.Variable, self.var)
```

```
obj = Classy()  
obj.var = 3  
obj.method()
```

2 3

Metody:

- dostęp innych metod obiektu również uzyskuje się poprzez self

Programowanie obiektowe w Pythonie

Metody

```
class Classy:
    def other(self):
        print("other")
    def method(self):
        print("method")
        self.other()

obj = Classy()
obj.method()
```

method
other

Metody:

- metoda `__init__` (czyli konstruktor) może być tylko jedna
- obowiązkowo musi mieć parametr `self`
- dodatkowe parametry są opcjonalne
- nie może zwracać wyniku
- nie może być wołana ani z obiektu, ani z klasy, ani ze świata zewnętrznego (wyjątkiem jest wołanie z konstruktora podklasy)

Programowanie obiektowe w Pythonie

Metody

```
class Classy:  
    def __init__(self,value):  
        self.var = value  
  
obj1 = Classy("object")  
print(obj1.var)
```

object

Metody:

- poza podanymi wymaganiami `__init__` zachowuje się jak każda metoda/funkcja, w szczególności w zakresie sposobów przekazywania argumentów i domyślnych wartości parametrów

Programowanie obiektowe w Pythonie

Metody

```
class Classy:
    def __init__(self,value = None):
        self.var = value

obj1 = Classy("object")
obj2 = Classy()
print(obj1.var)
print(obj2.var)
```

object
None

Metody:

- uprywatnienie metody wymaga rozpoczęcia jej nazwy od `__`

Programowanie obiektowe w Pythonie

Metody

```
class Classy:
    def visible(self):
        print("visible")
    def __hidden(self):
        print("hidden")

obj = Classy()
obj.visible()
try:
    obj.__hidden()
except:
    print("failed")
obj._Classy__hidden()
```

visible

failed

hidden

Własności:

- zarówno klasa, jak i każdy jej obiekt mają własność `__dict__`, który pozwala zapoznać się z ich zawartością
- pamiętać jednak należy, że oba słowniki mają inną zawartość

Programowanie obiektowe w Pythonie

Metody

```
class Classy:
    Variable = 1
    def __init__(self):
        self.var = 2
    def method(self):
        pass
    def __hidden(self):
        pass
```

```
obj = Classy()
print(obj.__dict__)
print(Classy.__dict__)
```

```
{'var': 2}
{'__module__': '__main__', 'Variable': 1, '__init__': <function
Classy.__init__ at 0x7f3a04f96c80>, 'method': <function
Classy.method at 0x7f3a04f96d08>, '__Classy__hidden': <function
Classy.__hidden at 0x7f3a04f96d90>, '__dict__': <attribute
'__dict__' of 'Classy' objects>, '__weakref__': <attribute
'__weakref__' of 'Classy' objects>, '__doc__': None}
```

Własności:

- każda klasa (ale żaden obiekt) ma własność `__name__`, zawierający łańcuch z nazwą klasy

Programowanie obiektowe w Pythonie

Metody

```
class Classy:  
    pass  
  
print(Classy.__name__)  
obj = Classy()  
print(type(obj).__name__)  
# uwaga - poniższa instrukcja spowodowałaby podniesienie wyjątku  
# print(obj.__name__)
```

Classy

Classy

Własności :

- każda klasa (ale żaden obiekt) zawiera krotkę `__bases__`, która przechowuje klasy (nie nazwy klas – klasy!), które są bezpośrednimi klasami danej klasy
- porządek klas w krotce odpowiada hierarchii dziedziczenia
- klasa bez jawnie podanej nadklasy jest podklasą klasy `object`

Programowanie obiektowe w Pythonie

Metody

```
class Super1:
    pass
class Super2:
    pass
class Sub(Super1, Super2):
    pass
def printbases(cls):
    print('( ',end='')
    for x in cls.__bases__:
        print(x.__name__,end=' ')
    print(')')
printbases(Super1)
printbases(Super2)
printbases(Sub)
```

```
( object )
( object )
( Super1 Super2 )
```