



Faculty of Geodesy
and Cartography

WARSAW UNIVERSITY OF TECHNOLOGY

INFORMATYKA GEODEZYJNA - WYKŁADY/ĆWICZENIA, ROK AKAD. 2020-2021

WYK. 2: PYTHON - ZMIENNE, OBIEKTY I REFERENCJE

Kinga Węzka

kinga.wezka@pw.edu.pl

Katedra Geodezji i Astronomii Geodezyjnej

**Warsaw University
of Technology**





1. Zmienne, obiekty i referencje
2. Typy danych (obiektów) w Pythonie
 - LICZBY: Wartości logiczne (boolowskie) – bool (True, False)
 - LICZBY: Liczby całkowite – int()
 - LICZBY: Liczby zmiennoprzecinkowe – float()
 - LICZBY: Liczby zespolone - complex()
 - KOLEKCJE – SEKWENCJE
 - KOLEKCJE – SEKWENCJE: lista – list()
 - KOLEKCJE – SEKWENCJE: krotka – tuple()
 - KOLEKCJE – SEKWENCJE: metody (indeksowanie i wycinki)
 - KOLEKCJE – ODWZOROWANIA: słownik – dict()
 - KOLEKCJE - ZBIORY: zbiory – set()
 - INNE TYPY: "Nie-liczby" (ang. not-a-number)
3. Instrukcje przypisania
4. Podstawowe operatory i kolejność działań
5. Podsumowanie
6. Materiały dodatkowe do ćwiczeń
 - Metody wbudowanych typów zmiennych
 - Pułapki typów wbudowanych



Zmienne, obiekty i referencje



- **Zmienna** to *konstruktor programistyczny*, który przechowuje różnego rodzaju dane niezbędne do działania programu. Podczas działania programu zmienna może zmieniać swoje wartości (jak wskazuje nazwa).
- Zmienne są przechowywane w pamięci komputera. Najprościej można je sobie wyobrazić jako pudełka, do których coś wkładamy. W pudełkach zapisywane są pewne informacje, a aby otrzymać do nich dostęp (do zmiennych) musimy odwołać się do nich po nazwie.



- Python nie posiada zmiennych w standardowym rozumieniu, każda zmienna w Pythonie jest **referencją do obiektu**, a wartość zmiennej jest **obiektem** o odpowiednim typie.
- Pojęcie zmienna i odniesienie do obiektu będzie wykorzystywane zamiennie.
- Zmienne są **typowane dynamicznie** – wynikowi działań automatycznie jest przypisany typ zmiennej z największą precyzją.

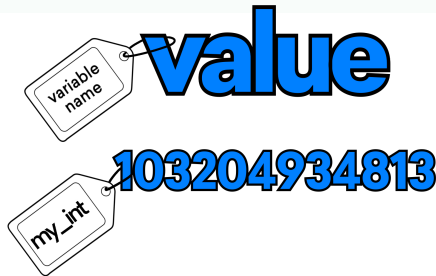
Instrukcja przypisania (ang. *assignment statement*):

```
zmienna (identyfikator)= wartość(obiekt)
```

```
my_int = 103204934814
```

- `my_int` jest nazwą zmiennej (identyfikatorem) i zawiera **referencję do obiektu**. W Pythonie wszystkie wartości są przekazywane przez referencję;
- `=` łączy w pamięci odniesienie wraz z obiektem;
- `103204934814` jest **obiektem** (dana wartość lub wyrażenie). To obiekt posiada typ (np. integer).

Typy są dynamiczne, co oznacza, że zmiennej nie deklarujemy typu. W momencie nadania wartości zmienna będzie przechowywać dany typ, dopóki, w trakcie działania kodu nie zdecydujemy się przypisać do niej innej wartości.



W Pythonie zmienne nie mają typu!
Wartości zmiennych mają typ!



Jak prawidłowo nazywać zmienne w Pythonie? – PEP 8 – Style Guide for Python Code

- Pierwszym znakiem powinna być mała litera alfabetu, ASCII lub Unicode – co oznacza, że można używać znaków alfabetu łacińskiego, ale również polskich znaków (nie polecane!)
- Do rozdzielania nazwy zmiennej składającej się z dwóch członów zaleca się użycie podkreślenia (`_`) np. `my_variable`
- Nazwy zmiennych są *case-sensitive* – tzn. że ważna jest wielkość liter. Dlatego `myvariable`, `myVariable` i `MyVariable` to 3 różne nazwy zmiennych.
- Nie używamy nazewnictwa jednoliterowego dla niejednoznacznie identyfikowanych liter np. `I` lub `l` lub `O` i innych bardzo podobnych do siebie znaków `1`, `0`, które w niektórych czcionkach są trudne do rozróżnienia.

Więcej w dok. **Propozycje Ulepszeń w Pythonie** ang. *Python Enhancement Proposals*.
PEP 8 – Style Guide for Python Code: <https://www.python.org/dev/peps/pep-0008/>



Typy danych (obiektów) w Pythonie



Klasyfikacja wbudowanych typów obiektów (ang. *built-in types*) w Pythonie:

<https://realpython.com/pointers-in-python/>

■ Liczby [immutable]:

- `bool`: wartości logiczne (boolowskie)
- `int` : liczby całkowite
- `float`: liczby zmiennoprzecinkowe
- `complex`: liczby zespolone

■ Sekwencje :

- `str`: [immutable] ciągi tekstowe
- `tuple`: [immutable] krotka
- `list`: [mutable] listy
- `bytes`: [immutable] bajty
- `bytearray`: [mutable] tablice bajtowe

■ Odwzorowania (ang. *mapping*):

- `dict`: [mutable] słownik, czyli tablica asocjacyjna

■ Zbiory:

- `set`: [mutable] zbiór
- `frozenset`: [immutable] zbiory zamrożone

■ Inne:

- `inf` (ang. *infinity*)
- `nan`
- `None`
- `NotImplemented`
- `Ellipsis` (np. `[1:2, ..., 0]`)



Wartości logiczne (boolowskie)

`bool` - typ logiczny, typ boolowski (ang. boolean), posiada dwie wartości: (`True` i `False`) – można je przedstawiać następująco: prawda (`True`, 1, +) i fałsz (`False`, 0, -)

■ Utworzenie obiektu (zmiennej):

```
1 a = True
2 b = False
```

■ Konwersja typu:

```
1 a = bool(0) # False
```

■ Przykłady prawdy i fałszu:

```
1 "mielonka" True
2 "" # False
3 [] # False
4 {} # False
```

```
1 1 # True
2 0.0 # False
3 None # False
```



Liczby całkowite: int()

Liczby całkowite (ang. *integer*) są typami o nieograniczonej wielkości (ogranicza je jedynie pamięć komputera).

■ Utworzenie obiektu (zmiennej):

```
1 a = 5
2 b = int(6)
3 c = int(3.3)
```

■ Konwersja typu:

```
1 a = int(3.45) # > 3
2 b = int('4') # > 4
```



Liczby zmiennoprzecinkowe - float()

Liczby zmiennoprzecinkowe (ang. *floating number*) są typami niezmiennymi (ang. *immutable*) o podwójnej precyzji (ang. *double precision*), podobnie jak C, C++ czy Java, precyzja ta wynosi 15-17 miejsc po przecinku. Do obliczeń wymagających większej precyzji należy skorzystać ze zmiennej `Decimal` z biblioteki standardowej `Decimal`.

■ Utworzenie obiektu (zmiennej):

```
1 a = 5.232
2 a = float(23)
3 c = 62.1E+6      # 62.1 * 10^6 = 62100000.0
```

■ Konwersja typu:

```
1 a = float(3) # 3.0
2 a = float('3') # 3.0
```



- Wbudowane (ang. *build-in*) operatory arytmetyczne dostępne dla liczb zmiennoprzecinkowych i całkowitych:

```
1 +   # dodawanie
2 -   # odejmowanie
3 *   # mnożenie
4 **  # potęgowanie (2**2 = 4)
5 /   # dzielenie -> float
6 //  # dzielenie z zaokrągleniem do najbliższego int w dół (15//2=7)
7 %   # reszta z dzielenia (11%4 = 3; -11%5 = 4 = 3*(-5)+4 )
```

- Wbudowane (ang. *build-in*) funkcje arytmetyczne dostępne dla liczb zmiennoprzecinkowych i całkowitych

```
1 abs(a)      # wartość absolutna
2 pow(x,y)    # x do potęgi y
```



Liczby zespolone - complex()

- $(1+2j)$ - liczby zespolone (ang. *complex number*)

```
1 a = (2+0j)
2 a = complex(2)
```

- Konwersja typu:

```
1 a = complex(3) # 3.0
```

- Wbudowane metody dla typu:

```
1 imag      # wyświetlenie jednostki urojonej
2 real      # wyświetlenie jednostki rzeczywistej
3 conjugate # sprzężenie zespolone
```



- Sekwencja to **uporządkowana pod względem pozycji** kolekcja obiektów. **Zachowują porządek** zawieranych elementów od lewej do prawej strony. Elementy te są przechowywane i pobierane zgodnie z ich pozycją.
- W Pythonie sekwencjami są:
 - **str** – łańcuchy znaków,
 - **list** – listy,
 - **tuple** – krotki,
- Dzielą one wspólne operacje na sekwencjach, takie jak: **indeksowanie**, **konkatenacja** czy **wycinki** (ang. *slice*), jednak każdy ma również specyficzne dla danego typu metody

Łańcuchy znaków: `str()`

Łańcuchy znaków (ang. *strings*) wykorzystywane są do przechowywania informacji tekstowych lub dowolnych zbiorów bajtów. Obiekt `str` to **niemutowalna** (tzn. niezmienna) **sekwencja** znaków, do których dostęp można uzyskać za pomocą **indeksowania**. Znaki to wartości kodów z odpowiedniego zestawu znaków (UTF)

■ Utworzenie łańcucha znaków (ang. *string*):

```
1 In [1] : s = 'witamy na wykladach w sali:'
2 In [2] : x = '315'
3 In [3] : print(type(s), type(x))
4 Out [3] : <class 'str'> <class 'str'>
```

■ Konwersja typu:

```
1 In [2] : b = str(2)
2 In [3] : type(b)
3 Out [3] : <class 'str'>
```



Na łańcuchach znaków nie można wykorzystywać podstawowych operatorów arytmetycznych (jak dla typów liczbowych). Łańcuchy znaków obsługują:

- **konkatenacja** (ang. *concatenation*): "+", łączącą dwa łańcuchy znaków w jeden):

```
1 In [1]: a = 'hello'
2 In [1]: b = 'world'
3 In [2]: c = a + b
4 Out [2]: 'hello world'
```

- **powtórzenie** (ang. *repetition*): "*", utworzenie nowego łańcucha znaków poprzez wielokrotność:

```
1 In [3]: d = 3*'tak'
2 Out [3]: 'taktaktak'
```

- **dołączanie** : "+=", sekwencyjne dołączanie elementu do łańcucha (w pętlach):

```
1 In [2]: a += '!' # równoważny zapis: a = a + '!'
2 Out [2]: 'hello!'
```




Inne metody/funkcje/operacje dostępne dla łańcucha znaków:

■ sprawdzenie długości łańcucha:

```
1 In [1]: s = 'hello world' # utworzenie zmiennej s typu string
2 In [2]: len(s) # zwraca długość napisu s (liczba bajtów/liczba znaków)
3 Out [2]: 11
```

■ test przynależności, sprawdzenie obecności łańcucha w łańcuchu:

```
1 In [3]: "w" in s # zwraca True jeśli string "w" jest częścią s
2 Out [3]: True
```



Lista to kolekcja, którą można porównać do tablic w innych językach programowania. Ważną cechą list jest to, że mogą przechowywać różne typy danych. Rozmiar ogranicza procesor.

■ Tworzenie listy:

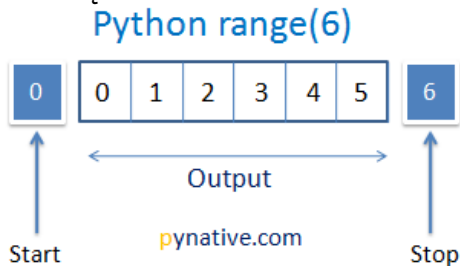
```
1 a = [] # stworzy pusta liste
2 b = list() # konwersja typu: utworzenie pustej listy
3 c = [1,2,3,4] # stworzy liste o wartosciach 1,2,3,4
4 d = 10*[3] # stworzy liste 10 liczb o wartosci 3
5 e = list(range(20)) # stworzy liste 20 liczb od 0 do 19
```

■ Przypisanie tworzy referencję a nie kopię (kopię wykonujemy poprzez użycie `b = a[:]`)

```
1 In [1]: a = [1,2,3,4,5]
2 In [2]: b = a # to jest referencja, kopię utworzy polecenie: b = a[:]
3 In [3]: b[1] = 'tekst' # lista jest mutowalna, zmiana 1 elementu
4 Out[3]: [1, 'tekst', 3, 4, 5]
5 In [5]: print(a)
6 Out[5]: [1, 'tekst', 3, 4, 5] # referencja wiec a również zmienione
```



- Generator listy: funkcja build-in (przydatna w tworzeniu list): **range(start, stop, step)**. Funkcja **range()** nie zwraca listy zawierającej liczby, zwraca **iterator**, aby uzyskać listę należy rzutować generator na listę.



- Przykład: zdefiniowanie listy od 1 do 20 z interwałem 2: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

```
1 In [1] : a = range(1, 20, 2)
2 Out [1] : range(1, 20, 2)
3 In [2] : a = list(range(1, 20, 2)) # rzutujemy aby wyświetlić listę
4 Out [2] : [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```



- W związku z tym, że lista może przechowywać różne typy danych. Możemy zdefiniować zmienną typu lista która będzie zawierać inne listy (tzw. **listy wielopoziomowe** lub **listy w liście**) np.:

```
1 nowa_lista = [[1,2,3], [4,5,6,7,8], ['a', 'b', 'c']]
```

- odwoływanie się do poszczególnych wartości wygląda następująco: `nowa_lista[x][y]`, gdzie `[x]` jest elementem listy głównej, `[y]` jest elementem listy na pierwszym poziomie zagnieżdżenia.

```
1 nowa_lista[2][1] # zwróci pierwszą wartość drugiej listy czyli 'b'
```



Krotka: tuple()

Krotki (ang. tuples) są bardzo podobne do list z tą różnicą, że są typem **niezmiennym** i deklaracja zmiennych zapisywana jest w nawiasach zwykłych, a nie kwadratowych. Również mogą przechowywać wiele typów danych jednocześnie. **Krotkę można zastosować jako klucz w słowniku (listy nie można zastosować jako klucz w słowniku).**

■ Deklaracja i inicjalizacja krotki:

```
1      krotka = ()                                # utworzenie pustej krotki
2      krotka = tuple()                          # utworzenie pustej krotki
3      krotka = (1, 2, "Jacek", "ma")           # utworzenie krotki z elementami
```



- Odwoływanie się do elementów krotki – indeksacja – podobnie jak list i str:

```
1 In [1] : krotka = (1, 2, "Jacek", "ma") # utworzenie krotki o  
      nazwie 'krotka'  
2 In [2]: krotka_liczb = krotka[:2] # Out [2]: (1,2)  
3 In [3]: krotka_stringow = krotka[2:]  
4 Out [3]: ('Jacek', 'ma')
```

- Pakowanie krotki (ang. *tuple packing*) (często w definicji funkcji)

```
1 In [6]: t = 5, 6, 7  
2 In [7]: x, y, z = t # przypisanie > otrzymamy x = 5, y = 6, z = 7
```

- Metody dla krotek:

```
1 t.count(war) # zwraca ilość wystąpień wartości 'war' w krotce  
2 t.index(war) # zwraca index wartości 'war'. Zwraca błąd jeśli  
      wartość nie istnieje
```



Indeksowanie i wycinki są metodami wspólnymi dla wszystkich typów sekwencyjnych Pythona: **(str(), list(), tuple())**:

- Elementy sekwencji (indeksy) listy liczymy od 0!
- Prawa strona zakresu wycinania jest niedomknieta!

To znaczy że np. dla sekwencji typu lista $L = [1, 2, 3, 4, 5]$ odwołanie się $L[1:4]$ zwróci listę $[2, 3, 4]$

```
1 a[-1]      # ostatni element -- nie będący sekwencją
2 a[-1:]     # ostatni element -- będący sekwencją
3 a[-2:]     # ostatnie dwa elementy -- będący sekwencją
4 a[:-2]     # cała sekwencja poza ostatnimi dwoma elementami
5 a[::-1]    # tworzy nowa sekwencje w odwrotnej kolejności
6 a[::2]     # stworzy nowa sekwencje z elementami dla indeksów parzystych
7 a[1::2]    # stworzy nowa sekwencje z elementami dla indeksów nieparzystych
```



Słownik: dic()

Słowniki to tablica mieszająca lub inaczej **tablica z haszowaniem**, którą można porównać do tablic asocjacyjnych znanych z innych języków programowania. Słowniki przechowują pary **klucz:wartość** i właśnie po kluczu odbywa się wyszukiwanie.

- Kluczem w słowniku może być każdy niezmienny typ danych np., string lub liczba. Kluczem może być również krotka, jeżeli zawiera typy niezmiennicze (string, liczba, krotka).
- **Klucze w słowniku są unikalne**
- Pary elementów nie są uporządkowane w kolejności, w której zostały dodane.

- Tworzenie słownika: pusty słownik:

```
1 Dict1 = {}           # utworzy pusty słownik o nazwie Dict1
2 Dict2 = dict()       # utworzy pusty słownik o nazwie Dict2
```




- `dict()` - słowniki - tworzymy "wkładając" pary klucz:wartość w nawiasy klamrowe, oddzielając je przecinkiem:

```
1 słownik = {}  
2 słownik = {'jeden': 1, 'dwa': 2, 'trzy': 3} # zalecana metoda  
3  
4 słownik = dict([('jeden', 1), ('dwa', 2), ('trzy', 3)])
```

- Dodawanie elementu do słownika:

```
1 Dict['key1']=3
```

- Usuwanie elementu słownika:

```
1 del Dict['key1']
```



■ Wybrane operacje i metody słownikowe:

```
1 "a" in słownik          # sprawdzenie czy klucz jest w słowniku czy nie
2 słownik.keys()          # wypisanie wszystkich kluczy
3 słownik.values()        # wypisanie wszystkich wartości
4 "a" in słownik.keys()   # czy klucz występuje w słowniku
```

■ Dodanie elementu do słownika

```
1 słownik["cztery"] = 4   # dodanie elementu (klucz= "cztery" i
                           wartość=4)
```

■ Dodanie elementu do słownika

```
1 słownik.setdefault(klucz,błąd)
```

Argument "błąd" jest opcjonalny – zwraca wartość elementu korespondującego z kluczem, jeśli klucza nie ma w słowniku zwraca wartość błąd i wstawia do słownika parę klucz:błąd (np. `słownik.setdefault('nowy_klucz', [])` - zawsze doda nowy klucz z wartościami nowej listy nawet jeśli ten klucz nie istniał w słowniku)



Zbiór: set()

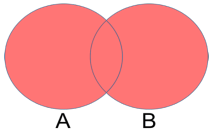
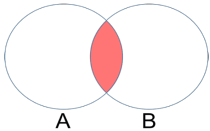
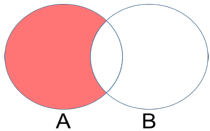
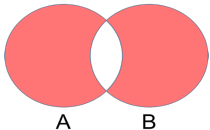
Zbiór (ang. set) kolekcja danych która obsługuje również matematyczne operacje z teorii zbiorów: suma, przecięcie, różnica oraz różnica symetryczna.

- **Elementy w zbiorze są unikalne**, duplikaty w zbiorze są eliminowane.
- **Elementy w zbiorze nie są uporządkowane**. W zależności od implementacji elementy mogą być sortowane za pomocą skrótu przechowywanej wartości, ale podczas korzystania z zestawów należy zakładać, że elementy są uporządkowane losowo.
- Definiowanie zbioru:

```
1 In [1] : zbior = set('nowy zbior')
2 Out [1] : {'w', ' ', 'i', 'o', 'r', 'y', 'n', 'z', 'b'}
```

Zbiór można zastosować jeżeli chcemy wyeliminować duplikaty z listy. Wystarczy rzutować listę na zbiór (czyli zamienić listę na zbiór), a następnie na wyjściu należy ponownie listę rzutować w odwrotną stronę.



Union		$A \cup B$, is the set of all values that are a member of A , or B , or both.
Intersection		$A \cap B$, is the set of all values that are members of both A and B .
Difference		$A \setminus B$, is the set of all values of A that are not members of B
Symmetric Difference		$A \triangle B$, is the set of all values which are in one of the sets, but not both.



- `inf` (ang. *infinity*)
- `NaN` (Not a Number), w Pythonie jest typem `float`
- `None` (None Type)
- `NotImplemented`
- `Ellipsis` – wielokropek np. `[1, 2, ..., 6]`

None vs NaN

```
1 type(math.nan) #> float
2 type(None)     #> NoneType
3 math.nan == math.nan #> False
4 math.nan is math.nan #> True
5 math.isnan(a)     #> True
6 None == None #> True
7 None is None #> True
```



Instrukcje przypisania



Instrukcje przypisania w Pythonie:

Cele zawierają referencje do obiektów, a wyrażenia generują obiekty (Lutz, 2015, p.75-78).

■ **proste przypisanie**

```
1 # cel = wyrażenie
2 a = 67.7556           # > wynik: a = 67.7556
```

■ **przypisanie wielocelowe** (ang. multiple-target) – służy do przypisania tego samego wyrażenia do każdego z celów.

```
1 # cel1 = cel2 = wyrażenie
2 a = b = 13.5           # > wynik: a = 13.5 i b = 13.5
```

■ **przypisanie krotek** – łączy w pary cele z wyrażeniami, od lewej do prawej.

```
1 # cel1, cel2 = wyrażenie1, wyrażenie2
2 a, b = 6.78, 14.7       # > wynik: a = 6.78 i b = 14.7
```



Instrukcje przypisania w Pythonie:

Cele zawierają referencje do obiektów, a *wyrażenia* generują obiekty (Lutz, 2015, p.75-78).

- **przypisanie z aktualizacją** — jest skrótem dla połączenia operacji matematycznej (np. dodawania, mnożenia itp) z przypisaniem. Przypisanie często wykorzystywane w pętlach.

```
1 # cell += wyrażenie
2 a += 1                                # > równoważne do zapisu: a = a + 1
```

 $X += Y$ $X \&= Y$ $X -= Y$ $X *= Y$ $X \wedge= Y$ $X /= Y$ $X \%= Y$ $X <=<= Y$ $X **= Y$ $X = Y$ $X >>= Y$ $X // = Y$

Dodatkowo format ten pozwala na zastosowanie działań w miejscu dla typów mutowalnych (np. wyrażenie `list1 += list2` automatycznie wywołuje metodę `list1.extend(list2)` zamiast wolniejszej operacji **konkatenacji** implikowanej przez operator `+`).



Instrukcje przypisania w Pythonie:

Cele zawierają referencje do obiektów, a *wyrażenia* generują obiekty (Lutz, 2015, p.75-78).

- **przypisania sekwencji** – które służy do przypisywania komponentów dowolnej sekwencji (bądź innego obiektu iterowalnego) do odpowiadających im celów, od lewej do prawej.

```
1 # cel1, cel2, ... = obiekt_iterowalny-o-odpowiedniej-długości
2 a, b, c, d = [1, 2, 3, 4] # > wynik: a = 1, b = 2, c = 3 i d = 4
```

- **rozszerzone instrukcje przypisania sekwencji** (Python 3.x) - zmienna celu poprzedzona gwiazdką (*) pobiera wszystkie niedopasowane elementy sekwencji.

```
1 # cel1, *cel2, ... = obiekt_iterowalny-o-odpowiedniej-długości
2 a, *b, c = [1, 2, 3, 4] # > wynik: a = 1, b = [2, 3] i c = 4
```



Podstawowe operatory i kolejność działań



1	+	<i># dodawanie</i>
2	-	<i># odejmowanie</i>
3	*	<i># mnożenie</i>
4	**	<i># potęgowanie</i>
5	/	<i># dzielenie -> float</i>
6	//	<i># dzielenie z zaokrągleniem do najbliższego int w dół</i>
7	%	<i># reszta z dzielenia</i>



```
1 x == y # równe co do wartosci
2 x != y # x nie jest równe y
3 x > y  # x jest większe niż y
4 x < y  # x jest mniejsze niż y
5 x >= y # x jest większe niż y lub równe y
6 x <= y # x jest mniejsze niż y lub równe y
```

Zwracają wartość: True/False



```

1  x or y      # Logiczna operacja OR (lub)
2  x and y     # Logiczna operacja AND (i)
3  not x       # Logiczna negacja
    
```

Przykłady:

```

1  In [1]: a = ''
2  In [2]: bool(a)
3  Out[2]: False
4  In [3]: b = 1
5  In [4]: bool(b)
6  Out[4]: True
7  In [5]: b and a
8  Out[5]: ''
9  In [6]: a or b
10 Out[6]: 1
    
```



```
1 is # tożsamość obiektów, porównuje wynik funkcji id() - adres w pamięci  
2 is not # sprawdza czy obiekty nie są tożsame
```

Przykłady:

```
1 In [1]: a = 7  
2 In [2]: b = 7.0  
3 In [3]: a == b  
4 Out[4]: True  
5 In [5]: a is b  
6 Out[5]: False
```

- Operator równości `==` sprawdza równość co do wartości. Python wykonuje test równości, porównując rekurencyjnie wszystkie zagnieżdżone obiekty (Lutz, 2011, p.286) .
- Operator tożsamości `is` sprawdza identyczność obiektów. Python sprawdza, czy dwa obiekty są tak naprawdę jednym (to znaczy znajdują się pod jednym adresem w pamięci).



```
1 in      # czy obiekt jest zawarty w innym obiekcie (np. wartość w liście)
2 not in  # czy obiekt nie jest zawarty w innym obiekcie
```

Przykłady:

```
1 In [1]: 'a' in 'ala'
2 Out[1]: True
3 In [2]: 'a' not in 'ala'
4 Out[2]: False
```

Testy przynależności wykorzystywane są często **dla sekwencji lub kolekcji** (ciągi tekstowe, krotki, listy) w celu sprawdzenia czy dana wartość znajduje się w zbiorze.



Gdy wyrażenie zawiera więcej niż jeden operator, kolejność wyznaczania zależy od użytych operatorów. Dla operatorów matematycznych **w Pythonie stosowana jest kolejność obowiązująca w matematyce** (Downey, 2016):

- **Nawiasy okrągłe mają najwyższy priorytet** (są wyznaczane jako pierwsze) i mogą posłużyć do wymuszenia wyznaczania wartości wyrażenia w żądanej kolejności, np. $2 * (3 - 1) = 4$ oraz $(1 + 1) ** (5 - 2) = 8$. Nawiasy stosowane są również w celu zwiększenia czytelności, np. $(minute * 100) / 60$, nawet gdy nie zmieniają wyniku.
- **Potęgowanie ma następny w kolejności priorytet**, czyli, $1 + 2 ** 3 = 9$ (a nie 27), $2 * 3 ** 2 = 18$ (a nie 36).
- **Mnożenie i dzielenie mają wyższy priorytet niż dodawanie i odejmowanie**, czyli $2 * 3 - 1 = 5$ (a nie 4), $6 + 4 / 2 = 8$ (a nie 5).
- **Operatory o takim samym pierwszeństwie są przetwarzane od lewej do prawej strony** (z wyjątkiem potęgowania). A zatem w wyrażeniu $degrees / 2 * pi$ dzielenie jest wykonywane jako pierwsze, a wynik mnożony jest przez wartość pi . Aby podzielić przez $2pi$ można użyć nawiasów lub wyrażenia w postaci $degrees / 2 / pi$.



1	<code>or</code>	<i># Logiczne OR (lub)</i>
2	<code>and</code>	<i># Logiczne AND (i)</i>
3	<code>not x</code>	<i># Logiczne NOT (nie)</i>
4	<code>in, not in</code>	<i># Testy przynależności</i>
5	<code>is, is not</code>	<i># Testy tożsamości</i>
6	<code><, <=, >, >=</code>	<i># Operatory porównania</i>
7	<code>!=, ==</code>	<i># Operatory równości</i>
8	<code> , ^, &</code>	<i># Bitowe: OR (lub), XOR (różnica symetryczna), AND (i)</i>
9	<code><<, >></code>	<i># Bitowe Przesunięcia</i>
10	<code>+, -</code>	<i># Dodawanie (konkatenacja) i odejmowanie (różnica)</i>
11	<code>*, %, /, //</code>	<i># Mnożenie, dzielenie, reszta z dzielenia</i>
12	<code>+x, ~x</code>	<i># Identyczność, negacja</i>
13	<code>~x</code>	<i># Bitowe NOT (nie)</i>
14	<code>**</code>	<i># Potęgowanie !!!</i>
15	<code>x[idx], x[i:k:k]</code>	<i># Indeksowanie i wycinanie z sekwencji (ang. slicing)</i>

Kolejność stosowania operatorów podana od najniższego priorytetu (Lutz, 2015, p.18)



Jeśli nie jesteśmy w stanie określić kolejności operatorów, stosujemy nawiasy okrągłe, aby pierwszeństwo było oczywiste. (Lutz, 2015, p.18)

<https://data-flair.training/blogs/python-operator-precedence/>



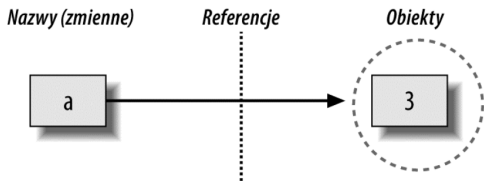
Podsumowanie



`a = 3`

Instrukcje przypisania w Pythonie odzwierciedlają:

1. Utworzenie obiektu reprezentującego wartość **3**.
2. Utworzenie zmiennej **a**, o ile jeszcze nie istnieje.
3. Połączenie zmiennej **a** z nowym obiektem 3.



Zmienna **a staje się referencją do obiektu **3**.**

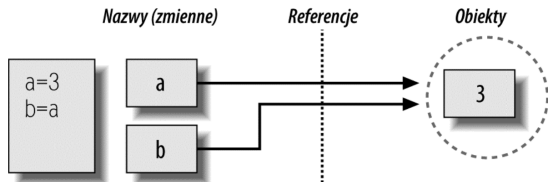
Wewnętrznie zmienna jest wskaźnikiem do miejsca w pamięci zajmowanego przez obiekt i utworzonego przez wykonanie wyrażenia literału **3** (Lutz, 2011).

Za każdym razem, gdy zmienne są używane korzystają z referencji, Python automatycznie podąża połączeniem między zmienną a obiektem:

- **Zmienne** są wpisami w tabeli systemowej z miejscem na łącze do obiektów.
- **Obiekty** to fragmenty przydzielonej pamięci z ilością miejsca potrzebnego do umieszczenia wartości, które reprezentują.
- **Referencje** to połączenia pomiędzy zmiennymi a obiektami (referencja to rodzaj powiązania zastosowanego jako wskaźnik w pamięci).



Nazwy (zmiennie) i obiekty po przypisaniu $b = a$. Zmienna b staje się referencją do obiektu 3. Wewnętrznie **zmienna jest tak naprawdę wskaźnikiem do miejsca w pamięci** zajmowanego przez obiekt i utworzonego przez wykonanie wyrażenia literału 3.

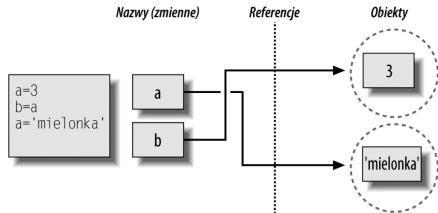


```
1      In [1]: a = 3
2      In [2]: b = a
3
4      In [3]: id(a)
5      Out[3]: 94490718053472
6
7      In [4]: b
8      Out[4]: 3
9
10     In [5]: id(b)
11     Out[5]: 94490718053472
```

Funkcja `id()` jest funkcją z biblioteki wbudowanej (ang. *build-in*) - zwraca numer referencji (tożsamość) obiektu.



Nazwy (zmiennne) i obiekty po przypisaniu `a = 'mielonka'`. Po wykonaniu wyrażenia z literałem `'mielonka'` zmienna `a` staje się referencją do nowego obiektu (miejsca w pamięci), natomiast zmienna `b` nadal odwołuje się do oryginalnego obiektu - liczby 3. Ponieważ to przypisanie nie jest modyfikacją obiektu 3 w miejscu, modyfikuje ono jedynie zmienną `a`, a nie `b`



```
1      In [1]: a = 3
2      In [2]: b = a
3      In [3]: a = 'mielonka'
4
5      In [4]: id(a)
6      Out[4]: 140445064605040
7
8      In [5]: a
9      Out[5]: 'mielonka'
10
11     In [6]: b
12     Out[6]: 3
13
14     In [7]: id(b)
15     Out[7]: 94490718053472
```



- Nazwy (zmiennne) nie mają typów.
- Nazwy (zmiennne) są referencją do obiektu.
- Typy powiązane są z obiektami, a nie ze zmiennymi.
- Przypisanie tworzy referencje, nie kopie.
- Typowanie zmiennych jest dynamiczne, nie ma konieczności deklaracji typu, a obiekty są uwalniane automatycznie.
- Typy dynamiczne są podstawą polimorfizmu (wielopostaciowość), czyli podstawowej cechy paradygmatu obiektowego.

- Przypisanie tworzy referencje, a nie kopię ([Lutz, 2011](#), p.283).
- W Pythonie obiekty (wartości zmiennych, a nie zmienne), posiadają typ.
- Python jest językiem z dynamicznym typowaniem zmiennych.



Materiały dodatkowe do ćwiczeń



Przykłady omawianych typów zmiennych

Deklaracja typu zmiennej w Pythonie nie jest wymagana, wystarczy podanej nazwie przypisać jakąś wartość za pomocą operatora przypisania “=”. Zmiennym często przypisujemy również wartości za pomocą wyrażeń np. $(a = 1 + 3 - 4)$, czyli działań arytmetycznych lub logicznych.

1	<code>a = False</code>	<code># bool</code>	<code># niezmienna (immutable)</code>
2	<code>b = 1</code>	<code># int</code>	<code># niezmienna (immutable)</code>
3	<code>c = 1.0</code>	<code># float</code>	<code># niezmienna (immutable)</code>
4	<code>d = "znienna napisowa"</code>	<code># string</code>	<code># niezmienna (immutable)</code>
5	<code>f = (1, 2)</code>	<code># tuple</code>	<code># niezmienna (immutable)</code>
6	<code>e = [1, 2, 3, 4, 5]</code>	<code># list</code>	<code># zmienna (mutable)</code>
7	<code>g = {'key1': 3, 'key2': 5.2}</code>	<code># dictionary</code>	<code># zmienna (mutable)</code>
8	<code>h = set('nowy zbiór')</code>	<code># set</code>	<code># zmienna (mutable)</code>



Pomoc dla metod i funkcji określanie typu obiektu

- wyświetlenie wszystkich metod danego obiektu - polecenie:
`dir(nazwa_obiektu)`
- wyświetlenie pomocy dla danej funkcji - polecenie:
`help(nazwa_funkcji)`
- określenie typu obiektu realizowane jest za pomocą funkcji: `type(nazwa_obiektu)`



Wybrane funkcje **standardowej** biblioteki **math** do działania na typach float (**import** math)

- `math.trunc()`: zwraca liczbę całkowitą (eliminuje część dziesiętną).
- `math.ceil()`: zwraca najmniejszą liczbę całkowitą większą niż podana liczba (zaokrąglenie w górę).
- `math.floor()`: zwraca największą liczbę całkowitą mniejszą niż podana liczba całkowita.

W przypadku liczb dodatnich `math.trunc()` i `math.floor()` dają ten sam wynik. Ale dla liczb ujemnych `math.floor()` zaokrągla w dół, a `math.trunc()` zaokrągla w górę.

```
1 In [1]: math.trunc(8.9)
2 Out[1]: 8
3 In [2]: math.floor(8.9)
4 Out[2]: 8
5 In [3]: math.trunc(-8.9)
6 Out[3]: -8
7 In [4]: math.floor(-8.9)
8 Out[40]: -9
```



```
1 s.capitalize()      # zmienia pierwszą literę na dużą
2 s.center(długość)   # Centruje napis w polu o podanej długości
3 s.count(sub)        # zlicza wystąpienie podciągu sub w napisie s
4 s.join(t)           # łączy napisy listy t używając s jako separatora
5 s.replace(old, new) # zastępuje stary podciąg nowym
6 s.split(separator)  # dzieli na podstawie separatora, (lista stringów)
7 s.strip()           # usuwa początkowe i końcowe białe znaki
8 s.rstrip()          # usuwa końcowe białe znaki
9 s.lstrip()          # usuwa początkowe białe znaki
10 s.isalnum()         # sprawdza czy znaki są znakami alfanumerycznymi
11 s.isdigit()         # sprawdza czy wszystkie znaki są cyframi
12 s.islower()         # sprawdza czy wszystkie litery są małe
13 s.isspace()         # sprawdza czy wszystkie znaki są białymi znakami
14 s.isupper()         # sprawdza czy wszystkie litery są duże
15 s.encode(coding)    # zwraca zakodowaną wersję napisu ('utf-8', 'ascii')
```



Sekwencje specjalne dla łańcuchów

Literały (jednostka leksykalna) łańcuchowe mogą zawierać **sekwencje specjalne** reprezentujące bajty o specjalnym znaczeniu:

- `\\` : dosłowny znak `\`
- `\'` : apostrof: `'`
- `\"` : cudzysłów: `"`
- `\a` : kod BEL - sygnał akustyczny (BIP!) lub wizualny (błysk)
- `\b` : kod BS ('backspace') - cofa kursor o jedną pozycję, pozwala nadpisać ostatni znak
- `\f` : kod FF ('formfeed') - przejście do następnej strony
- `\n` : kod LF ('linefeed') - znacznik końca linii w strumieniu tekstowym
- `\r` : kod CR ('carriage return') - cofa kursor do początku aktualnej linii
- `\t` : kod HT ('horizontal tab') - tabulator
- `\v` : kod VT ('vertical tab') - działa analogicznie jak HT, ale w kierunku pionowym
- `\N{nazwa}` : znak o danej nazwie symbolicznej w tabeli Unicode



```
1 s = [1, 2, 3, 4] # definiuje listę s
2 list(s)          # konwertuje sekwencję s na listę
3 len(s)           # podaje długość listy (ilość elementów)
4 s.append(x)       # dodaje nowy element x na końcu s
5 s.extend(t)       # dodaje nową listę t na końcu s
6 s.insert(i,X)     # wstawia na pozycję i wartość X
7 s.count(x)        # zlicza wystąpienie x w s
8 s.index(x)        # zwraca najmniejszy indeks i, gdzie s[i] == x
9 s.pop([i])        # zwraca i-ty element i usuwa go z listy
10 s.remove(x)       # odnajduje x i usuwa go z listy s
11 s.reverse()      # odwraca w miejscu kolejność elementów s
12 s.sort()         # sortuje wartości listy np od najmniejszej do największej
```



- `append` - dodaje obiekt na końcu

In[1]:

```
1 x = [1, 2, 3]
2 x.append([4, 5])
3 print (x)
```

Out1:

```
1 [1, 2, 3, [4, 5]]
```

- `extend` - rozbudowuje listę poprzez dodawanie elementów (łączy pierwszą listę z innym obiektem nie koniecznie listą)

In[2]:

```
1 x = [1, 2, 3]
2 x.extend([4, 5])
3 print (x)
```

Out2:

```
1 [1, 2, 3, 4, 5]
```



```
1 d = {'jeden': 1, 'dwa': 2, 'trzy': 3} # definiuje słownik d
2 d.clear() # usuwa wszystkie pary klucz:wartość ze słownika d.
3 d.copy() # zwraca d, płytkie kopiowanie słownika d.
4 d.values() # zwraca listę wartości słownika.
5 d.keys() # zwraca listę kluczy.
6 d.items() # zwraca listę elementów (klucz, wartość).
7 d.iteritems() # iterator nad elementami słownika d.
8 d.iterkeys() # iterator nad kluczami słownika d.
9 d.itervalues() # iterator nad wartościami słownika d.
10 d.get(k,d) # zwraca d[k] jeśli k jest indeksem w słowniku, w
    przeciwnym razie zwraca d.
11 d.has_key(klucz1) # zwraca True jeśli słownik d ma klucz - klucz1, w
    przeciwnym wypadku False.
12 d.popitem() # zwraca parę (klucz, wartość) i usuwa ze słownika
    (jeśli słownik jest pusty "zwraca" - KeyError.
```




- 1 `d.fromkeys(S,v)` # parametr *v* jest opcjonalny – tworzy słownik *d* z
kluczami z *S* i wartościami równymi *v*. (np. `data = {}`; `data =`
`data.fromkeys(range(2),[])`)
- 2
- 3 `d.pop(klucz[,blad])` # parametr *blad* jest opcjonalny – zwraca wartość
elementu korespondującego z kluczem i usuwa go ze słownika, jeśli
klucza nie ma w słowniku zwraca wartość *blad*, a jeśli nie została
podana "zwraca" *blad* – `KeyError`.
- 4
- 5 `d.setdefault(klucz,blad)` # parametr *blad* jest opcjonalny – zwraca
wartość elementu korespondującego z kluczem, jeśli klucza nie ma w
słowniku zwraca wartość *blad* i wstawia do słownika parę `klucz:blad`.
- 6
- 7 `d.update(E[,**F])` # nie zwraca wartości, aktualizuje `Dict` parami
`klucz:wartość` ze słowników podanych jako parametry.



```
1 >>> dic1 = {'a': 10, 'b': 5, 'c': 3}
2 >>> dic2 = {'d': 6, 'c': 4, 'b': 8}
3 # aktualnie dostępna metoda łączenia słowników:
4 >>> {**dic1, **dic2}
5 {'a': 10, 'b': 8, 'c': 4, 'd': 6}
6 # Od 3.9 Python w operacjach ze słownikami wykorzystuje metody zbiorów
7 >>> dic1 | dic2
8 {'a': 10, 'b': 8, 'c': 4, 'd': 6}
9 >>> dic2 | dic1
10 {'d': 6, 'c': 4, 'b': 8, 'a': 10}
11 # pozwala na modyfikację w miejscu:
12 >>> dic1 = {'a': 10, 'b': 5, 'c': 3}
13 >>> dic1 |= dic2
14 >>> dic1 # >
15 {'a': 10, 'b': 8, 'c': 4, 'd': 6}
```



- Zwraca zbiór będący połączeniem zbiorów A i B.

```
1 A | B # lub tak: A.union(B)
```

- Zapisuje wszystkie elementy tablicy B do zbioru A.

```
1 A |= B # lub tak: A.update(B)
```

- Zwraca zbiór będący przecięciem (częścią wspólną) zbiorów A i B

```
1 A & B # lub tak: A.intersection(B)
```

- Pozostawia w zbiorze A tylko elementy należące do zbioru B.

```
1 A &= B # lub tak: A.intersection_update(B)
```

- Elementy zawarte w A, ale nie zawarte w B.

```
1 A - B # lub tak: A.difference(B)
```

- Usuwa wszystkie elementy B ze zbioru A.

```
1 A -= B # lub tak: A.difference_update(B)
```



- Elementy należące do A lub B, ale nie do obu zbiorów jednocześnie.

```
1 A ^ B    # lub tak: A.symmetric_difference(B)
```

- Zapisuje w A symetryczną różnicę zbiorów A i B.

```
1 A ^= B    # lub tak: A.symmetric_difference_update(B)
```

- Zwraca wartość true, jeśli A jest podzbiorem B.

```
1 A <= B    # lub tak: A.issubset(B)
```

- Zwraca prawdę jeśli B jest podzbiorem A.

```
1 A >= B    # lub tak: A.issuperset(B)
```

- Równoważne dla wyrażenia : $A \leq B$ and $A \neq B$

```
1 A < B
```

- Równoważne dla wyrażenia : $A \geq B$ and $A \neq B$

```
1 A > B
```



- Istnieją dwie metody usunięcia elementu z zestawu: `discard` i `remove`. Ich zachowanie różni się tylko w przypadku, gdy usunięty element nie znajdował się w zbiorze. W tym przypadku metoda `discard` nic nie robi, a metoda `remove` zgłasza wyjątek `KeyError`.

```
1 In [1]: klasa = {"Marek", "Janek", "Ania", "Ewa", "Marek", "Ania"}
2 In [2]: klasa.remove("Ewa")
3 In [3]: klasa.discard("Ewa")
```

- Metoda `pop` usuwa jeden losowy element ze zbioru i zwraca jego wartość. Jeśli zbiór jest pusty, metoda ta generuje wyjątek `KeyError`.

```
1 In [4]: klasa = {"Marek", "Janek", "Ania", "Ewa", "Marek", "Ania"}
2 In [5]: klasa.pop()
```

- Możesz wykonać konwersję (rzutowanie) zbioru do listy za pomocą listy funkcji.

```
1 In [6]: klasa = {"Marek", "Janek", "Ania", "Ewa", "Marek", "Ania"}
2 In [7]: lista = list(klasa)
3 Out [7]: ['Ania', 'Janek', 'Ewa', 'Marek'] # duplikaty usunięte
```



■ Przypisanie tworzy referencje, nie kopie (Lutz, 2011, p.292)

```
1 In [1]: L = [1, 2, 3]
2 In [2]: M = ['X', L, 'Y']    # Osadzenie referencji do L
3 In [3]: M
4 Out[3]: ['X', [1, 2, 3], 'Y']
5
6 In [4]: L[1] = 0             # Wprowadzi zmiany również w M
7 In [5]: M
8 Out[2]: ['X', [1, 0, 3], 'Y']
```

```
1 In [1]: L = [1, 2, 3]
2 In [2]: M = ['X', L.copy(), 'Y'] # Osadzenie kopii do L: (L[:])
3 In [3]: L[1] = 0                # Zmiana tylko L, nie M
4 In [4]: M
5 Out[4]: ['X', [1, 2, 3], 'Y']
```



- Powtórzenie dodaje jeden poziom zagłębienia (Lutz, 2011, p.293). Powtórzenia w sekwencjach przypominają dodanie sekwencji do samej siebie jakąś liczbę razy.

```
1 In [1]: L = [4, 5, 6]
2 In [2]: X = L * 4           # Jak [4, 5, 6] + [4, 5, 6] + ...
3 In [3]: Y = [L] * 4        # [L] + [L] + ... = [L, L, ...]
4 In [4]: X
5 Out[4]: [4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
6 In [5]: Y
7 Out[5]: [[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

- Uwaga na cykliczne struktury danych. Jeśli kolekcja obiektów zawiera referencję do samej siebie, nazywana jest **obiektem cyklicznym**.

```
1 In [1]: L = ['Gaal']
2 In [2]: L.append(L)
3 In [3]: L
4 Out[3]: ['Gaal', [...]] # [...] - zamiast nieskończonej pętli
```



- Typów niezmiennych nie można modyfikować w miejscu! Typy niezmiennie to: bool, int, float, string, tuple.

```
1 In [1]: T = (1, 2, 3)
2 In [2]: T[2] = 4      # Błąd!
3
4 In [2]: T = T[:2] + (4,)
5 Out[2]: (1, 2, 4)
```




Dla chetnych



```
1 In [1] : a = 3.7 + 1.1
2 In [2] : b = 4.8
3 In [3] : print(a == b)
4 Out [3] : False
```

```
1 In [1] : print(a)
2 Out [1] : 4.8000000000000001
```

Komputery "liczą" w systemie binarnym, tj. o podstawie 2 (0,1), a my liczymy w systemie dziesiętnym, czyli o podstawie 10 (0,1,2...8,9). Różnica polega na tym, że nie wszystkie ułamki o podstawie 10 da się przedstawić za pomocą ułamka o podstawie 2.

- Dla nas, w **systemie dziesiętnym**: $3.7 = 3 + 7/10$
- **A w systemie binarnym**:
 $3.7 = 3 + 1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128 + 1/256 + 0/512 + 0/1024 + 1/2048 + \dots$
- Ile byśmy tych ułamków nie dodali to za "..." nie wyjdzie liczba równa 3.7
- Dlatego komputery nie są w stanie poprawnie liczyć na ułamkach dziesiętnych (poza "sztuczkami" istniejącymi w każdym sensownym języku (Python) typ Decimal).

Przydatne narzędzie: IEEE 754 Converter:

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>



Aby z liczby dziesiętnej uzyskać liczbę dwójkową należy dzielić daną liczbę przez 2, wyniki kolejnych dzielen zapisać w słupku, a reszty z dzielen zapisać po prawej stronie za kreską. Kolejne dzielenia wykonujemy do momentu gdy wynik z dzielenia mniejszy od 1. Przeliczenie liczby 173 z systemu dziesiętnego (decymalny - dec) do systemu dwójkowego (binarny - bin).

1	173:2	reszta:	1
2	86:2	reszta:	0
3	43:2	reszta:	1
4	21:2	reszta:	1
5	10:2	reszta:	0
6	5:2	reszta:	1
7	2:2	reszta:	0
8	1:2	reszta:	1

Wynik: 173 (w systemie dziesiętnym) = 10101101 (w systemie binarnym)

Wystarczy przepisać uzyskane reszty z dzielen od dołu do góry i mamy wynik: 10101101



Aby zamienić liczbę z systemu dwójkowego na dziesiętny należy najpierw przypomnieć sobie jak są tworzone liczby w ww. systemach - jaka liczba jest ich podstawą. **Podstawą w systemie dwójkowym jest liczba 2**, w systemie szesnastkowym liczba 16, a w systemie dziesiętnym liczba 10. Aby przeliczyć liczbę z systemu dwójkowego na dziesiętny należy:

$$x \cdot 2^n + \dots + x \cdot 2^5 + x \cdot 2^4 + x \cdot 2^3 + x \cdot 2^2 + x \cdot 2^1 + x \cdot 2^0$$

- Załóżmy, że chcemy przeliczyć z systemu dwójkowego na dziesiętny liczbę: **10101101**
- W powyższym wzorze w miejsca "x" wstawiamy na odpowiednie (kolejne) pozycje kolejne cyfry z przeliczanej liczby:

$$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 1$$

- Widząc zależność że każda następna potęga liczby 2 jest od swojego poprzednika dokładnie dwukrotnie większa:

$$1128 + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = \mathbf{173}$$



- Aby zamienić część ułamkową z systemu dziesiętnego na system dwójkowy należy dziesiętny należy najpierw mnożyć kolejną część dziesiętną przez 2. Następnie część całkowitą z kolumny **obliczenia** wpisujemy jako bit, a część ułamkową poddajemy kolejnemu mnożeniu przez 2. Operacje powtarzamy dopóty, część dziesiętna nie wynosi 0.

Część ułamkowa	Obliczenia	Część całkowita (bit)	Reszta
0.375	$(0.375 \times 2 = \mathbf{0.750})$	0	$0.750 - 0 = 0.750$
0.750	$(0.750 \times 2 = \mathbf{1.5})$	1	$1.5 - 1 = 0.5$
0.5	$(0.5 \times 2 = \mathbf{1.0})$	1	$1 - 1 = 0.0$

- Otrzymujemy:

$$(0.375)_{10} = (0.011)_2$$

- Przeliczenie ułamka dziesiętnego z systemu dwójkowego do binarnego:

$$(0.011)_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = \frac{1}{4} + \frac{1}{8} = \frac{2}{8} + \frac{1}{8} = \frac{3}{8} = 0.375$$



- Niestety nie zawsze każdy ułamek naturalny da się zamienić na ułamek dziesiętny. Tak samo nie każdy ułamek dziesiętny da się z łatwością zamienić na ułamek binarny ze skończoną liczbą bitów. Powodem jest jego okresowość. **Jeśli ułamek jest nieskończenie okresowy to nie da się go zapisać dokładnie w postaci dziesiętnej**

$$\frac{1}{3} = (0.3333333333...)_{10} = (0.(3))_2$$

- Dotyczy to również konwersji ułamka na ułamek w postaci binarnej, np. $(0.6)_{10}$

Część ułamkowa	Obliczenia
0.6	$(0.6 \times 2 = \mathbf{1.2})$
0.2	$(0.2 \times 2 = \mathbf{0.4})$
0.4	$(0.4 \times 2 = \mathbf{0.8})$
0.8	$(0.8 \times 2 = \mathbf{1.6})$
0.6	$(0.6 \times 2 = \mathbf{1.2})$
0.2	$(0.2 \times 2 = \mathbf{0.4})$
0.4	$(0.4 \times 2 = \mathbf{0.8})$
0.8	$(0.8 \times 2 = \mathbf{1.6}) \downarrow$
0.6	$(0.6 \times 2 = \mathbf{1.2})$
⋮	⋮



Typy danych dostępne w bibliotekach standardowych (ang. *standard library*)

- decimal (<https://docs.python.org/3/library/decimal.html>),
- fractions (<https://docs.python.org/3.1/library/fractions.html>),
- collections (<https://docs.python.org/3/library/collections.html>)

Formaty wyświetlania: `str()` i `repr()`

Obie funkcje konwertują dowolne obiekty na ich reprezentacje łańcuchowe.

- `repr()` - zwraca kanoniczną* reprezentację ciągu obiektu, zwraca wyniki wyglądające tak, jakby były one kodem (domyślne wyświetlanie z sesji interaktywnej)
- `str()` - dokonuje konwersji na format bardziej przyjazny dla użytkownika, jeśli jest dostępny

```
1 In [1]: num = 1.0/3.0
2 In [2]: print(str(num)) # zwraca w przyjaznej formie
3 Out [2]: '0.3333333333333333'
4 In [3]: print(repr(num)) # zwraca kanoniczną reprezentację
5 Out [3]: '0.33333333333333331'
```

W informatyce ***postać kanoniczna** oznacza takie przedstawienie, w którym każdy obiekt ma swoją unikatową reprezentację. W ten sposób można łatwo sprawdzić równość dwóch obiektów poprzez sprawdzenie równości ich postaci kanonicznych.



A. Downey. *Myśl w języku Python*. 2016.

M. Lutz. *Python. Wprowadzenie*. Helion, 2011.

M. Lutz. *Python, Leksykon kieszonkowy*. Helion, 2015.

ME AFTER 10 LINES OF CODING



Enough For Today!

Dziękuję za uwagę

Kinga Wężka

Gmach Główny PW – pok 38/6

kinga.wezka@pw.edu.pl