Moduł 2 - Podstawowe typy danych

1. Operatory.

Zanim omówione zostaną typy danych, warto poznać kilka operatorów, które w powiązaniu ze zmiennymi są często używane.

Listing 1

```
# operatory arytmetyczne

# operator dodawania
print(1 + 2)
# operator odejmowania
print(1 - 2)
# operator mnożenia
print(1 * 2)
# operator dzielenia z resztą
print(1 / 2)
# operator dzielenia bez reszty (dzielenie całkowite)
print(1 // 2)

# pamiętajmy o kolejności operacji arytmetycznych
wynik = 1 + 2 * 3 / 4.0

# operatory przypisania
zmienna = "wartość" # przypisanie wartości do zmiennej
# są też skrócone postacie operatorów przypisania w połączeniu z innymi
operatorami
suma = 10
suma += 1
# to samo co
suma = suma + 1
# podobnie możemy używać operatorów -, *, /, //, **, % i operatorów bitowych
(zachęcam do poczytania w dokumentacji)


# modulo czyli reszta z dzielenia
reszta = 12 % 5
# operator potęgowania
kwadrat = 5 ** 2
szescian = 5 ** 3

# operacje na zmiennych znakowych (string)
full_name = "Kowalski" + " " + "Jan"

# tak też można
spam = "SPAM " * 10
print(spam)
```

```python
# operatory porównania
liczba1 = 1
liczba2 = 2
print(liczba1 > liczba2)
print(liczba1 < liczba2)
print(liczba1 <= liczba2)
print(liczba1 >= liczba2)
print(liczba1 == liczba2)
print(liczba1 != liczba2)

# powyższe porównania zwrócą wartości typu bool czyli True lub False

prawda = True
falsz = False

# operatory logiczne
print(prawda and falsz) # koniunkcja logiczna
print(prawda or falsz) # alternatywa logiczba
print(not prawda) # negacja
print(not not prawda) # podwójna negacja
print(bool(prawda or falsz)) # użycie metody bool(), która jest tutaj wywołaniem
konstruktora klasy bool (więcej w kolejnych labach)

# operatory tożsamości (identity)
liczba = 1
liczba2 = liczba

print(liczba is liczba2)
print(liczba is not liczba2)

# operatory przynależności (membership)
lista = [1, 2, 3, 4]
print(1 in lista)
print(5 not in lista)

# operatory bitowe (zobacz:
https://docs.python.org/3.12/library/stdtypes.html#bitwise-operations-on-integer-
types)
x | y # logiczna alternatywy (ang. or) na x i y
x ^ y # logiczna alternatywa rozłączna na x i y
x & y # logiczna koniunkcja of x and y
x << n # x przesunięte w lewo o n bitów
x >> n # x przesunięte w prawo o n bitów
~x # przerzucanie bitów (negacja)

# operator przypisania := (nowość w Python 3.8.*)
# tutaj najpierw nowej zmiennej n przypisana zostaje wartość len(lista) a później
# dokonana zostaje ewaluacja tej zmiennej czy n > 10. Może to oszczędzić w
niektórych
# przypadkach nieco miejsca na deklarację zmiennej, jeżeli potrzebna jest tylko w
bardzo
# ograniczonym zakresie (ang. scope)
```

```
if (n := len(lista)) > 10:
    print(f"List is too long ({n} elements, expected <= 10)")
```

Python w wyrażeniach wykonuje działania w określonej kolejności:

1. najpierw `**`
2. następnie `*`, `/` oraz `%`
3. a dopiero na końcu `+` i `-`

Informacje o epratorach, rozbite niestety w wielu podrodziałach, znajdują się w dokumencie opisującym wbudowane typy danych języka Python: https://docs.python.org/3.12/library/stdtypes.html. Do tego dokumentu będziemy jeszcze wracać.

Bardziej szczegółowe informacje na temat priorytetów operatorów można znaleźć tu: https://docs.python.org/3.12/reference/expressions.html#operator-summary

W Pythonie jako fałsz traktowane są:

- liczba zero (`0, 0.0, 0j` itp.)
- `False`
- `None`
- puste kolekcje (`[], (), {}, set()` itp.)
- puste łańcuchy znakowe
- w Pythonie 2 – obiekty posiadające metodę `__nonzero__()`, jeśli zwraca ona `False` lub `0`
- w Pythonie 3 – obiekty posiadające metodę `__bool__()`, jeśli zwraca ona `False`

2. Typy liczbowe.

Dwa główne typy liczbowe Pythona to liczba całkowita oraz rzeczywiste, czyli **int** i **float**. Jest jeszcze typ **complex**, który służy do przechowywania wartości liczb zespolonych, ale zapoznanie się z informacjami na jego temat pozostawiam czytelnikowi.

Listing 2

```
calkowita = 5
rzeczywista = 5.6
rzeczywista = float(56)
# powyższy sposób to rzutowanie - konwersja jednego typu w inny o ile to możliwe
# poniżej kolejny przykład
liczba_str = '123'
liczba = int(liczba_str)
print(type(liczba))

# zmienne można również zadeklarować w jednej linii
a, b = 3, 4

# typ int w Pythonie nie ma odgórnego ograniczenia jeżeli chodzi o
# maksymalną wartość (32 czy 64 bity), a jedyne ograniczenie to ilość dostępnej
pamięci
```

```
# nieco inaczej jest w przypadku typu float
# maksymalna precyzja zależy od wersji interpretera i systemu
# operacyjnego, a szczegóły można znaleźć tu:
# 1. https://note.nkmk.me/en/python-sys-float-info-max-min/
# 2. https://docs.python.org/3/tutorial/floatingpoint.html
```

W przypadku liczb rzeczywistych można również określić precyzję, z jaką zostaną wyświetlone (ale nie przechwywane w pamięci), ale stosowny przykład znajduje się w kolejnym podrozdziale.

Ważnym aspektem pracy z dziesiętnymi liczbami zmiennoprzecinkowymi jest sposób ich zapisu w sytemie binarnym, który może powodować problemy pokazane na poniższym listingu.

Listing 3

```
from decimal import Decimal

print((0.1 + 0.2) == 0.3)  # kto by się spodziewał ?
print(round((0.1 + 0.2), 2) == round(0.3, 2))  # teraz lepiej
print((Decimal(0.1) + Decimal(0.2)) == Decimal(0.3))
print((Decimal('0.1') + Decimal('0.2')) == Decimal('0.3'))
```

Wyjaśnienie niezgodności wyników dwóch ostatnich funkcji **print()** z kodu powyżej można znaleźć w dokumentacji modułu **Decimal**: https://docs.python.org/3/library/decimal.html

Dla zainteresowanych problemem reprezentacji liczb zmiennoprzecinkowych w systemie dwójkowym odsyłam do dość obrazowo wyjaśnienia w artykule pod adresem http://www.samouczekprogramisty.pl/liczby-zmiennoprzecinkowe/.

3. Typ `str`. Wybrane właściwości.

> Dokumentacja: https://docs.python.org/3.12/library/stdtypes.html#str

Typ łańcuchowy jest chyba najbardziej powszechnym typem danych nie tylko w języku Python. Dane pobierane z plików, bez względu na ich postać są pierwotnie zapisane jako łańcuch znaków, a następnie mogą być skonwertowane na inny typ, np. numeryczny. Zapisanie danych do pliku wyjściowego również wymaga często ich rzutowania (zamiany typu) na typ `str`.

W etapie przygotowania i oczyszczania danych (ang. data cleaning, data mangling) bardzo często praca polega na wykorzystaniu metod i technik pracy z łańcuchami znaków.

Poniżej przykłady deklaracji zmiennej typu `str`.

Listing 4

```
artykul = """Recenzja "Władcy Pierścieni"."""
imie = 'Jan'
hobby = "piłka nożna"
```

```
# lub poprzez rzutowanie innego typu na typ str
liczba = 1000
# wykorzystanie konstruktora obiektu do stworzenia obiektu typu str z
# obiektu typu int
liczba_str = str(liczba)
```

Powyższy fragment to tylko przykład różnych metod deklaracji, w trakcie zajęć będą stosowane apostrofy.

Ciąg tekstowy w Pythonie to tablica znaków, co daje z miejsca wiele możliwości manipulacji i dostępu do składowych tego ciągu. Inna ważna cecha stringów to fakt, że po ich zadeklarowaniu nie możemy zmienić zadeklarowanych znaków ciągu, gdyż zmienne typu string są niemutowalne (ang. immutable). Oczywiście możemy nadpisać zmienną nową wartością, czyli zmienić wartość przez ponowne przypisanie.

Listing 5

```
imie = 'Kowalski'
nazwisko = 'Jan'

# string to tablica znaków więc możemy odwołać się do jej elementów
print(imie[0])

# indeks elementu możemy również określać jako pozycja od końca ciągu
print(imie[-1])

# można również pobrać fragment ciągu (slice) określając jako indeks
# element początkowy i końcowy. Zwróć uwagę na wartość tych indeksów.
print(imie[0] + imie[-2] + imie[4:6])
# można również określic tylko jeden z dwóch indeksów
# co oznacza od elementu o indeksie 3 do końca łańcucha
print(imie + nazwisko[3:])

# ogólna postać slice
# [start:stop:step]
# wartości poszczególnych parametrów slice'a są pomijalne, ale
# musimy zapisywać drukropki, które informują mechanizm o tym
# które parametry zostaną uzyte z ich domyślnymi wartościami
# sprawdź działanie poniższych przykładów
print(imie[::2])
print(imie[-2])
print(imie[:-4:-1])
print(imie[::-1])

# inny sposób złączania ciągów
print(imie + ' ' + nazwisko)

# Elementów ciągu nie można zmieniać więc poniższa instrukcja zwróci błąd.
# nazwisko[0] = "P"

# Potwierdzeniem tego, że ciąg tekstowy jej również obiektem jest możliwość
# wykonania na nim metod dla tego typu zdefiniowanych. Metoda count() zlicza
# ilość wystąpień danego ciągu w wartości przechowywanej przez zmienną.
```

```
print(imie.count('z'))
# Co ciekawe w Pythonie możemy wywoływać funkcje dla danego obiektu już podczas
deklaracji
# co na pierwszy rzut oka może wyglądać dość egzotycznie.
print('Jesteś szalona!'.count('a'))

# Potwierdzeniem niezmienności zadeklarowanego stringa może być również poniższy
kod
print(imie.lower())
print(imie)

# Aby zwrócić długość ciągu tekstowego należy posłużyć się wbudowaną funkcją len()
print(len(nazwisko))
```

Wybrane własności klasy `str`.

Klasa `str` zapewnia konstruktor, który zwraca postać łańcuchową przekazanego obiektu. Każdy obiekt w Pythonie posiada taką postać, chociaż czasem to co uzyskamy na wyjściu nie zawsze jest pomocne. Dzieje się to z powodu dziedziczenia innych obiektów po klasie `object`, która dostarcza między innymi magicznej metody `__str__()`.

Osoby zainteresowane zgłębianiem kodu źródłowego implementacji CPython odsyłam do: https://github.com/python/cpython

Technicznie przy wywołaniu `str(obiekt)` wywoływana jest metoda `__str__()` dla danego typu obiektu, a jeżeli tam nie została ona zaimplementowana to poprzez mechanizm dziedziczenia będzie kaskadowo poszukiwana metoda `__str__()` obiektu rodzica, aż do abstrakcyjnej ogólnej klasy `object`.

Listing 6

```
# czy będzie jakaś różnica w oknie konsoli ?
print(str(5))
print(5)
print(int(5).__str__())
```

Poniżej przedstawiony zostanie fragment kodu, a którym wykorzystane zostanie kilka wybranych metod dla klasy `str`. Dla pełnej listy odsyłam do oficjalnej dokumentacji.

Listing 7

```
# 1. wczytujemy dane ze standardowego wejścia (tu przechwytujemy wejście z
klawiatury w oknie konsoli)
sentence = input('Wpisz dowolne zdanie:\n')
# dane przechowywane są w postaci zmiennej typu str

# 2 - możemy usunąć ewentualne białe znaki na początku i końcu łańcucha.
sentence = sentence.strip()
# można ograniczyć się tylko do początku lub końca łańcucha używając odpowiednio
metod
```

```
# lstrip() i rstrip()

# 3 - Jedną z metod jest metoda split, która dzieli łańcuch na części, a każda z
nich jest
# zapisywana do listy w postaci łańcucha znaków bez uwzględnienia znaku
separatora.
# Łańcuch zostanie podzielony i zliczone zostaną słowa (zakładają, że spacje jest
separatorem)
words = sentence.split(' ')

"""
Można pominąć ' ' w tym przypadku i efekt będzie taki sam, ale to nie oznacza, że
domyślnie
jest to po prostu spacja.
Za dokumentacją:
If sep is not specified or is None, a different splitting algorithm is applied:
runs of consecutive whitespace are regarded as a single separator, and the result
will contain no empty strings at the start or end if the string has leading or
trailing whitespace. Consequently, splitting an empty string or a string
consisting of just whitespace with a None separator returns [].
"""

print('W podanym zdaniu jest ', len(words), ' słów.')

# 4 - możemy również sprawdzić, czy zdanie rozpoczyna się wielką literą
print('Czy zdanie rozpoczyna się wielką literą? -> ', words[0].istitle())

# 5 - lub czy zdanie rozpoczyna się znakiem alfabetycznym
print('Czy zdanie rozpoczyna się od litery? -> ',sentence[0].isalpha())

# 6 - teraz wczytamy inne wejście - oczekujemy podania liczby
num = input('Wpisz dowolną liczbę:\n')

print('Czy liczba jest liczbą całkowitą? -> ', num.isdigit())

# 7 - jak wprowadzone zostaną instrukcje warunkowe i pętle to będzie łatwiej :)
# sprawdzimy, czy wartość była podana jako docelowy float (separator . )
print('Czy liczba jest liczbą zmiennoprzecinkową? -> ',
num.replace('.','',1).isdigit())

# 8 - dopełnimy liczbę poprzedzającymi zerami do 5 pozycji
# aktualnie zadziała tylko dla liczby całkowitej
print(num.zfill(5))
```

4. Formatowanie łańcuchów znaków.

Formatowanie łańcuchów znaków polega na łączeniu wzorców tekstowych z wartościami innych typów. Wiemy już, że możemy to zrobić poprzez uzycie funkcji `print()` np. `print('Adam ma ', 23 ,' lata')` i nie jest to najgorszy pomysł. Ma jednak pewne wady, a mianowicie nie mamy tutaj zbyt wielu możliwości określenia formatu, w jakim wartość niebędąca łańcuchem znaków zostanie wypisana, np. z dokładnością do

n-znaków po przecinku, wyrównana do strony prawej lub lewej, itd. Z pomocą przychodzą techniki formatowania łańcuchów znaków przedstawione na listingu poniżej.

Listing 8

```
# formatowanie znane z Pythona 2.x
wyznanie = 'Lubię %s' % 'Pythona'
print(wyznanie)
wonsz = 'Python'
print('Lubię %sa' % wonsz)

print('Lubię %s oraz %sa' % ('Pythona', wonsz))

# %s oznacza, że w miejsce tego znacznika będzie podstawiany ciąg tekstowy
# %i - to liczba całkowita
# %f - liczba rzeczywista lub inaczej zmiennoprzecinkowa
# %x lub %X - liczba całkowita zapisana w formie szesnastkowej

print('Używamy wersji Python %i' % 3)
print('A dokładniej Python %f' % 3.9)
print('Chociaż lepiej to zapisać jako Python %.1f' % 3.9)
print('A kolejną główną wersją Pythona może być wersja %.4f' % 3.11111)
print('A może będzie to wersja %.1f ?' % 3.111)
print('A może jednak %.f ?' % 3.12)
wersja = 4
print('A %i w systemie szesnastkowym to %X' % (wersja, wersja))
print('A %i * %i szesnastkowo daje %X' % (wersja, wersja, wersja*wersja))

# Chociaż możliwości przy korzystaniu z mechanizmów powyżej są spore,
# to i kilka wad się również znajdzie. Trzeba pilnować zarówno liczby argumentów,
jak
# i ich kolejności. Konieczne jest powielanie tej samej zmiennej, jeżeli kilka
# razy jest wykorzystywana w formatowanym ciągu. Spójrzmy na inne możliwości.

print('Lubię %(jezyk)s' % {'jezyk': 'Pythona'})
print('Lubię %(jezyk)s a czy Ty lubisz %(jezyk)s ?' % {'jezyk': 'Pythona'})
# wadą jest dość duża ilość dodatkowego kodu do napisania, ale nazwy zmiennych
# w ciągu pozwalają na ich szybką identyfikację i wielokrotne wykorzystanie w
# dowolnej kolejności

# poniżej kolejny sposób
print('Lubię język {1} oraz {0}'.format('Java', 'Python'))

# w nowej wersji języka Python możliwe jest również odwoływanie się do elementów
kolekcji
# lub pól klasy
class Osoba:

    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko
```

```
jan = Osoba('Jan', 'Kowalski')

print('Tą osobą jest {0.imie} {0.nazwisko}'.format(jan))
```

W wersji 3.6 wprowadzono do języka Python pojęcie **„f-string”**, które nieco upraszcza formatowanie ciągów tekstowych. Jest to obecnie zalecana i szeroko stosowana metoda. Przykład poniżej.

Listing 9

```
# zapis jest skróconą postacią użycia funkcji .format()
imie = 'Marek'
print(f'Witaj {imie}!')

# zapis wewnątrz {} powoduje wykonanie działań, które tam zapisano
# odwołanie do zmiennej, funkcji itp a następnie zastosowanie formatu
# określonego opcjonalnie po symbolu : wewnątrz tego nawiasu.
# poniższy przykład: najpierw wykonana zostanie operacja {33.33 + 66.67
# a następnie zastosowany format .4f, który oznacza zapisanie liczby
# zmiennoprzecinkowej (float - stąd litera f) z dokładnością do 4 liczb po
przecinku
print(f'Wynik dodawania 33.33 oraz 66.67 to {33.33 + 66.67: .4f}')
print(f'Imię zapisane wielkimi literami to {imie.capitalize()}.')
```

Po więcej przykładów związanych z formatowaniem łańcuchów można udać się pod poniższe adresy:

1. https://docs.python.org/3/library/string.html#format-string-syntax
2. https://pyformat.info/
3. https://realpython.com/python-f-strings/

Listing 10

```
# \r jest sekwencją sterującą (tu powrót karetki), sprawdź co to oznacza
print('elo\r', end='')
print('zero\r', end='')

# sekwencje sterujące są opisane tu:
# https://docs.python.org/3.12/reference/lexical_analysis.html#escape-sequences

# gdybyśmy jednak chcieli wypisać na konsoli łańcuch reprezentujący zarezerwowany
# dla sekwencji sterującej możemy to zrobić wykorzystując łańcuch 'surowy' lub
sekwencję escape (\\)

print('elo\\r', end='')
print(r'zero\r', end='')

# a dlaczego się to może przydać ? wypisz linie poniżej
print('\sciezka\rok\numer\task')
print(r'\sciezka\rok\numer\task')
print('\\sciezka\\rok\\numer\\task')
```

```
# można też wypisać znaki niestandardowe z tablicy Unicode
print("\u2764")
print("\u2765")
print("\U0001F602")

# tablica unicode w przyjaznej formie: https://symbl.cc/en/unicode/table/
```

Ćwiczenia

1. Napisz fragment kodu, który wczyta trzy zmienne ze standardowego wejścia (np. funkcja input()):
    - linię danych rozdzielonych jakimś separatorem (spacja, średnik, itd.)
    - separator źródłowy
    - separator docelowy

Następnie zaimplementuj z użyciem metod `str.split()` oraz `str.join()` podzielenie danych wejściowych pierwszym separatorem, połączenie i wypisanie danych połączonych drugim separatorem. Czy można to zrobić prościej wykorzystując inne wbudowane metody?

2. Użyj funkcji `input()` aby pobrać łańcuch znaków z klawiatury i z użyciem wycinków (slice) wykonaj:

- podziel łańcuch na dwie części, w miarę możliwości równe, ale jeżeli długość łańcucha jest nieparzysta, np. 11 znaków to pierwszy ma długość 5, a drugi 6. Wyświetl te łańcuchy w oknie konsoli.
- wyświetl łańcuch składający się z co drugiego znaku licząc od końca łańcucha

3. Wyświetl na konsoli dowolny ciąg znaków i wykorzystaj wbudowane metody:

- `title()`,
- `capitalize()`,
- `zfill()`,
- `upper()`,
- `count()`,
- `center()`.

4. Wprowadź z klawiatury dowolny łańcuch znaków i zapisz go do zmiennej. Następnie bazując na przykładzie poniżej zapisz również wyniki dla metod `isalpha(), isascii(), isprintable(), istitle(), isupper()`. `wejscie = input() print(f"Łańcuch {wejscie} isdecimal: {wejscie.isdecimal()}")`.

5. Przejdź na stronę https://pyformat.info/, a następnie zapisz w oddzielnym pliku .py i wykonaj 5 wybranych przykładów formatowania ciągów oznaczonego jako „New”, których nie było w przykładach z tego podrozdziału (np. z wyrównaniem do prawej lub lewej strony, ilością pozycji liczby, znakiem, wypełnieniem spacji itp.). Przerób zaprezentowane tam przykłady na postać z użyciem f-string.

6. Wykorzystując **listing 10** wypisz na konsoli 10 wybranych znaków niestandardowych (np. litery z alfabetu greckiego, symbole walut - (funt, bitcoin)) wypisując jednocześnie jego kod z tablicy unicode.