Python w Elektronicznej Sieci #3: Wprowadzenie do programowania w Pythonie (2/3)

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2020-03-10

1 Napisy

Do tej pory używaliśmy zmiennych do przechowywania liczb i operowania na nich. Zmienne mogą również jako wartości przyjmować litery, słowa, a nawet całe zdania:

```
x = 'A'
a, b, c = 'Ala', "ma", " kota i psa"
d = """ ... a co ma ...
 "kotek"?"""
print(x, a[2])
print(c[1], c[-1], c[-3])
print(a + b)
print(3 * a)
print(a + " " + b + c + d)
```

```
A a
o a p
Alama
AlaAlaAla
Ala ma kota i psa ... a co ma ...
 "kotek"?
```

Zwróć uwagę na następujące rzeczy:

- Napisy muszą być otoczone pojedynczymi apostrofami lub podwójnym cudzysłowami (nie ma znaczenia, którą wersję wybierzemy), w przypadku napisów wieloliniowych używamy trzykrotnie apostrofu lub cudzysłowowa na początku i końcu napisu.
  Nie przypisane do żadnej zmiennej napisy wieloliniowe mogą być stosowane jako komentarze wieloliniowe.
- Przy użyciu liczby w nawiasie kwadratowym możemy poznać poszczególne litery napisu (***numeracja rozpoczyna się od 0***).
- Ujemny indeks oznacza odliczanie liter od końca napisu: ostatnia litera napisu `c` to `c[-1]`, przedostatnia to `c[-2]`, itd.
- Przy użyciu znaku dodawania możemy sklejać (*konkatenować*) napisy.
- Przy użyciu znaku gwiazdki możemy mnożyć napisy (czyli sklejać same ze sobą).

Innymi przydatnymi operacjami na napisach jest sprawdzanie długości napisu poleceniem `len()` oraz wycinanie podnapisu przy użyciu dwukropka:

```
tekst = 'Python'
dlugosc = len(tekst)
print(dlugosc, tekst[2:5], tekst[3:], tekst[:3])
```

```
6 tho hon Pyt
```

W powyższym przykładzie:

- komenda `tekst[2:5]` zwraca podnapis od znaku nr 2 (**włącznie**) do znaku nr 5 (**wyłącznie**),
- komenda `tekst[3:]` zwraca podnapis od znaku nr 3 (**włącznie**) do końca,
- komenda `tekst[:3]` zwraca podnapis od początku do znaku nr 3 (**wyłącznie**).

Podobnie jak w `range()` możemy podać trzeci argument określający przedział czyli krok. Pozwala to na wybieranie co n-tego znaku z napisu, zarówno zaczynając od początku jak i końca:

```
tekst = '123456789'
dlugosc = len(tekst)
print(tekst[::2], tekst[1::2])
print(tekst[::-1], tekst[::-3])
print(tekst[::-1][::3], tekst[::3][::-1])
```

```
13579 2468
987654321 963
963 741
```

W powyższym przykładzie:

- komenda `tekst[::2]` zwraca co drugi znak,
- komenda `tekst[1::2]` zwraca co drugi znak od znaku nr 1,
- komenda `tekst[::-1]` zwraca napis od tyłu,
- komenda `tekst[::-3]` zwraca co 3 znak z napisu od tyłu (warto zauważyć że nie zawsze jest to równoważne wypisaniu napisu złożonego z co 3 znaku od tyłu).

1.1 Napis jako lista

Wszystkie listy, których do tej pory używaliśmy w pętli **for** były listami liczb. Okazuje się, że w Pythonie napisy mogą być traktowane jako lista, a dokładniej listą liter. Oznacza to, że po napisie można przejść przy użyciu pętli **for**, tak samo jak przechodziliśmy po liście liczb:

```
for l in 'Abc':
    print('litera', end = ' ')
    print(l)
```

```
litera A
litera b
litera c
```

1.1.1 Modyfikowalność napisów

Python pozwala odwoływać się do poszczególnych znaków w napisie jak do elementów listy, jednak nie pozwala na ich modyfikowanie:

```
s = "abcdefgh"
s[2] = "X"
print(s)
```

```
Traceback (most recent call last):
  File "python", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Zwróć uwagę na komunikat błędu, który został wyświetlony, podaje on informacji o tym co wywołało błąd (opis błędu) i w której linii programu on wystąpił. **Czytanie ze zrozumieniem komunikatów o błędach ułatwia naprawianie niedziałającego programu.**

Jeżeli zachodzi potrzeba modyfikowania napisu konkretnych znaków w napisie możemy użyć poznanej wcześniej metody uzyskiwania podnapisów:

```
s = "abcdefgh"
s = s[:2] + "X" + s[3:5] + s[6:]
print(s)
```

```
abXdegh
```

Powyższy przykład w miejsce znaku nr 2 wstawia napis "X" oraz usuwa znak nr 5 z napisu. Przy konieczności modyfikacji znak po znaku możemy użyć iteracji po napisie i budować nowy napis znak po znaku:

```
s, ns = "abcdefgh", ""
for z in s:
    if z in "cf":
        ns = ns + "X"
    else:
        ns = ns + z
print(ns)
```

```
abXdeXgh
```

1.2 Konwersje liczba – napis

Z punktu widzenia komputera liczba czy też element napisu, którym jest litera są pewną wartością numeryczną. Natomiast my do zapisu liczb używamy różnych systemów (np. dziesiętnego, czy też szesnastkowego). Domyślnie liczby wprowadzane do programu interpretowane są jako zapisane w systemie dziesiętnym, podobnie liczby uzyskiwane poprzez konwersję napisu przy pomocy funkcji `int()`. Możliwe jest jednak wprowadzanie liczb zapisanych w innych systemach liczbowych lub konwersja z napisu zawierającego liczbę — drugi, opcjonalny argument `int()` pozwala określić podstawę systemu z którego konwertujemy, zero oznacza automatyczne wykrycie w oparciu o prefix:

```
# szesnastkowo
h1, h2, h3 = 0x1F, int("0x1F", 0), int("1F", 16)
# oktalnie
o1, o2, o3 = 0o17, int("0o17", 0), int("17", 8)
# binarnie
b1, b2, b3 = 0b101, int("0b101", 0), int("101", 2)

print("",h1,o1,b1, "\n",h2,o2,b2, "\n",h3,o3,b3)
```

```
 31 15 5
 31 15 5
 31 15 5
```

Możliwe jest także konwertowanie wartości liczbowej na napis w określonym systemie liczbowym:

```
a, b = 3, 13
c = (a + b) * b
s = "(" + bin(a) + " + " + oct(b) + ") * " + hex(b) + " = " + str(c)
print( s )
```

1.3 Kodowania znaków

Python używa Unicode dla obsługi napisów, jednak przed przekazaniem napisu do świata zewnętrznego konieczne może być zastosowanie konwersji do określonej postaci bytowej (zastosowanie odpowiedniego kodowania). Służy do tego metoda `encode()` np.:

```
a = "ąąbcć ... ←↓→"
inUTF7 = a.encode('utf7')
inUTF8 =  a.encode() # lub a.encode('utf8')
print("'" + a + "' w UTF7 to: " + str(inUTF7) + ", w UTF8: " + str(inUTF8))
```

Zmienne typu 'bytes' oprócz przekazania na zewnątrz (np. zapisu do pliku lub wysłania przez sieć) mogą zostać także m.in. zdekodowane do napisu z użyciem metody `decode()` lub poddane dalszej konwersji np. kodowaniu base64:

```
print("zdekodowany UTF7: " + inUTF7.decode('utf7'))

import codecs
b64 = codecs.encode(inUTF8, 'base64')
print("napis w UTF8 po zakodowaniu base64 to: " + str(b64))
```

W powyższym przykładzie należy zwrócić uwagę na instrukcję **import**, która służy do załączania bibliotek pythonowych do naszego programu. W tym wypadku załączamy fragment standardowej biblioteki Pythona o nazwie codecs.

Base64 jest jednym z kodowań pozwalających na zapis danych binarnych w postaci ograniczonego zbioru znaków drukowalnych, co pozwala m.in. na osadzanie danych binarnych (np. obrazki) w plikach tekstowych (np. dokumenty html, pliki źródłowe programów).

1.3.1 Konwersja pomiędzy znakiem a jego numerem

Możliwe jest także konwertowanie pomiędzy liczbowym numerem znaku Unicode, a napisem go reprezentującym i w drugą stronę — służą do tego odpowiednio funkcje chr() i ord(). W ramach napisów można też użyć \uNNNN, gdzie NNNN jest (czterocyfrowym) numerem znaku lub po prostu umieścić dany znak w pliku kodowanym UTF8[1].

```
print(chr(0x221e) + " == \u221e == ∞")
print(hex(ord("∞")), hex(ord("\u221e")), hex(ord(chr(0x221e))) )
```

1.4 Wyrażenia regularne

W przetwarzaniu napisów bardzo często stosowane są wyrażenia regularne służące do dopasowywania napisów do wzorca który opisują, wyszukiwaniu/zastępowaniu tego wzorca. Do typowej, podstawowej składni wyrażeń regularnych zalicza się m.in. następujące operatory:

```
.       - dowolny znak
[a-z]   - znak z zakresu
[^a-z]  - znak z poza zakresu (aby mieć zakres z ^ należy dać go nie na początku)
^       - początek napisu/linii
$       - koniec napisu/linii

*       - dowolna ilość powtórzeń
?       - 0 lub jedno powtórzenie
+       - jedno lub więcej powtórzeń
{n,m}   - od n do m powtórzeń

()      - pod-wyrażenie (może być używane dla operatorów powtórzeń,
          a także dla referencji wstecznych)
```

1. Użyty w przykładzie symbol nieskończoności można uzyskać na standardowej polskiej klawiaturze pod Linuxem przy pomocy kombinacji AltGr + Shift + M

Python umożliwia korzystanie z wyrażeń regularnych za pomocą modułu `re`:

```
import re

y = "aa bb cc bb ff bb ee"
x = "aa bb cc dd ff gg ee"

if re.match(".*[dz]", y):
  print("y zawiera d lub z")

if re.match(".*[dz]", x):
  print("x zawiera d lub z")

if re.match(".* ([a-z]{2}) .* \\1", y):
  print("y zawiera dwa razy to samo")

if re.match(".* ([a-z]{2}) .* \\1", x):
  print("x zawiera dwa razy to samo")

# zastępowanie
print (re.sub('[bc]+', "XX", y, 2))
print (re.sub('[bc]+', "XX", y))

# zachłanność
print (re.sub('bb (.*) bb', "X \\1 X", y))
print (re.sub('.*bb (.*) bb.*', "\\1", y))
print (re.sub('.*?bb (.*) bb.*', "\\1", y))
```

```
x zawiera d lub z
y zawiera dwa razy to samo
aa XX XX bb ff bb ee
aa XX XX XX ff XX ee
aa X cc bb ff X ee
ff
cc bb ff
```

Zwróć uwagę na:

- Działanie funkcji `match`, która dopasowuje wyrażenie do początku napisu (czyli tak jakby zaczynało się od `^`).
- Odwołania wsteczne do pod-wyrażeń (fragmentów ujętych w nawiasy) postaci `\\x`, gdzie `x` jest numerem pod-wyrażenia.
- „*Zachłanność*” (ang. *greedy*) wyrażeń regularnych:
  - w pierwszym wypadku `bb (.*) bb` dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`,
  - w drugim przypadku gdy zostało poprzedzone `.*` dopasowało tylko `ff`, gdyż `.*` dopasowało najdłuższy możliwy fragment czyli `aa bb cc`,
  - w trzecim wypadku `bb (.*) bb` mogło i dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`, gdyż było poprzedzone niezachłanną odmianą dopasowania dowolnego napisu, czyli: `.*?`.

  Po każdym z operatorów powtórzeń (`. ? + {n,m}`) możemy dodać pytajnik (`.? ?? +? {n,m}?`) aby wskazać że ma on dopasowywać najmniejszy możliwy fragment, czyli ma działać nie zachłannie.

2 Zmienne i ich typy

2.1 Listy

Do tej pory listy traktowaliśmy głównie jako zbiór elementów po którym iterujemy. Zastosowanie list jest jednak znacznie szersze. Lista stanowi pewnego rodzaju kontener do przechowywania innych zmiennych, w którym elementy zorganizowane są na zasadzie określenia ich (względnej) kolejności. Lista może zawierać elementy różnych typów.

Na listach możemy wykonywać m.in. operacje modyfikowania, czy też usuwania jej elementów:

```
l = ["i", "C", 0, "M"]
l[0] = "I"
del l[2]
print(l)
```

```
['I', 'C', 'M']
```

W powyższym przykładzie widzimy:

- Modyfikację pierwszego elementu listy (`l[0] = "I"`), z użyciem odwołania poprzez numer elementu. Elementy list numerujemy od zera. Ujemne wartości oznaczają numerowanie od końca listy, czyli -1 jest ostatnim elementem listy, -2 przedostatnim, itd.
- Usunięcie trzeciego elementu listy (`del l[2]`). Powoduje to zmianę numeracji kolejnych elementów.

Jednak jeżeli chcemy modyfikować elementy listy iterując po niej, to konieczne jest iterowanie po indeksach (a nie jak dotychczas po wartościach):

```
for i in range(len(l)):
    print(l[i])
    l[i] = "q"
print(l)
```

```
I
C
M
['q', 'q', 'q']
```

Dzieje się tak gdyż przypisanie do zmiennej `x` jakiejś wartości w ramach konstrukcji `for x in lista:` modyfikuje tylko zmienną `x`, a nie element listy który został do niej pobrany.

2.1.1 Wybór podlisty

Możemy także tworzyć „podlisty” przy pomocy operatora zakresów w identyczny sposób jak to zostało opisane przy napisach, np. `ll[1::2]` zwróci listę złożoną z co drugiego elementu listy `ll` zaczynając od elementu o indeksie 1.

2.1.2 Lista jako modyfikowalny napis

Listy mogą też służyć jako narzędzie do modyfikowania napisów. W tym celu można skorzystać np. z listy złożonej z liter oryginalnego napisu:

```
s = "abcdefgh"
l = list(s)
l[2] = "X"
del(l[5])
s = "".join(l)
print(s)
```

```
abXdegh
```

2.2 Określanie typu zmiennej

Do tej pory poznaliśmy kilka typów zmiennych w Pythonie: liczby, napisy oraz listy. Poznaliśmy także metody konwersji pomiędzy niektórymi z typów (np. instrukcje `str()`, `int()`). Jeżeli chcemy dowiedzieć się jakiego typu jest dana zmienna możemy skorzystać z funkcji `type()`:

```
a, b, c = 1, 3.14, "Python"
print(a, type(a))
print(b, type(b))
print(c, type(c))
c = (a == 1)
print(c, type(c))
```

```
1 <class 'int'>
3.14 <class 'float'>
Python <class 'str'>
True <class 'bool'>
```

Zauważ że inny typ związany jest z liczbami całkowitymi, inny z rzeczywistymi, a jeszcze inny z wartościami logicznymi (`True`/`False`). Zauważ także, że zmienna może zmienić swój typ.

2.3 Obiektowość

Jak mogliśmy zauważyć przy sprawdzaniu typów zmiennych są one klasami. Związane z tym jest m.in. to iż posiadają one metody służące do operowania na nich. Opis danego typu wraz z dostępnymi metodami można obejrzeć przy pomocy polecenia `help()`, np. `help("list")`.

W przypadku list za pomocą metod tej klasy mamy możliwość wstawiania wartości na daną pozycję, sortowania i odwracania kolejności elementów:

```
l = ["i", "m"]
l.insert(1, "c")
print(l)
l.reverse()
print(l)
l.sort()
print(l)
```

```
['i', 'c', 'm']
['m', 'c', 'i']
['c', 'i', 'm']
```

Zwróć uwagę że sortowanie i odwracanie modyfikuje istniejącą listę a nie tworzy kopii.

2.4 Słowniki

Kolejnym użytecznym typem zmiennych w Pythonie są słowniki (zwane niekiedy *mapami* lub *tablicami asocjacyjnymi*). Podobnie jak listy służą do do przechowywania innych zmiennych. W odróżnieniu jednak od list w słownikach przechowywane są pary klucz - wartość, gdzie unikalny klucz służy do identyfikowania wartości.

```
slownik = { "bd" : "xx", 5: True, "a" : 11 }
for klucz in slownik:
    print (klucz, "=>", slownik[klucz])
```

```
a => 11
bd => xx
5 => True
```

Zauważ że zarówno klucz, jak i wartość mogą być dowolnego typu oraz że słownik nie zachowuje kolejności dodawania elementów.

Możliwe jest także sprawdzanie istnienia jakiegoś elementu w słowniku, usuwanie, dodawanie i zmienianie elementów słowniku, itd (zwróć także uwagę na inną metodę wypisywania słownika - poprzednio iterowaliśmy po kluczach, teraz po liście par klucz-wartość):

```
if "bd" in slownik:
    print ("jest element o kluczu 'bd'")
    del slownik['bd']
slownik[15] = True
slownik["a"] = "yy"
for k,v in m.items():
    print (k, "=>", v)
```

```
jest element o kluczu 'bd'
a => yy
15 => True
```

2.4.1 Sortowanie słownika

Jak już wspomnieliśmy słownik nie zachowuje porządku elementów. Jeżeli chcemy uzyskać posortowaną listę kluczy, wartości lub par klucz-wartość z słownika możemy skorzystać z funkcji `sorted()`. W przypadku par wywołanie będzie wyglądać następująco:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
lista = sorted( mapa.items() )
print(lista)
```

```
[('5', 3), ('a', 101), ('bd', 20)]
```

Zwróć uwagę, iż użyliśmy tej samej metody `items()`, z której korzystaliśmy do iterowania po parach klucz-wartość (dla listy samych kluczy lub wartości należy użyć w tym miejscu innej metody klasy `dict`). Zapewne zauważyłeś że sortowanie zostało przeprowadzone w oparciu o klucze, co jednak jeżeli chcielibyśmy posortować taką listę w oparciu o wartości? W takim przypadku możemy skorzystać z opcjonalnego argumentu funkcji `sorted()` o nazwie `key`, który przyjmuje funkcję mającą za zadanie na podstawie otrzymanego elementu listy (w tym wypadku pary klucz - wartość) zwrócić klucz sortowania:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
def k(x):
    return x[1]
lista = sorted( mapa.items(), key=k )
print(lista)
```

```
[('5', 3), ('bd', 20), ('a', 101)]
```

2.5 Zmienna, obiekt i referencja

W Pythonie każda zmienna jest nazwą wskazującą na jakiś obiekt w pamięci. Podobnie każdy element listy czy słownika wskazuje na jakiś obiekt[2]. Na jeden obiekt może wskazywać wiele zmiennych i/lub elementów innych obiektów (takich jak listy czy słowniki). Jeżeli zmienna nie ma na co wskazywać (np. został do niej przypisany wynik funkcji, która nie zwraca wartości) wskazuje na obiekt `None` (typu `NoneType`). Zatem na wszystkie zmienne pythonowe możemy patrzeć jak na referencje do obiektów istniejących gdzieś w pamięci.

Do uzyskania identyfikatora obiektu związanego z daną nazwą, lub elementem innego obiektu służy funkcja `id` (w przypadku standardowej implementacji Pythona jest to po prostu adres w pamięci).

2.5.1 Usuwanie i czas życia zmiennych

Instrukcja **`del`**, której używaliśmy już do usuwania elementów z listy lub słownika może być wykorzystana także do usuwania innych zmiennych. Należy jednak pamiętać iż w Pythonie usunięcie zmiennej nie wiąże się z natychmiastowym zwolnieniem zajmowanej przez nią pamięci z kilku powodów:

- na pojedynczy obiekt może wskazywać kilka zmiennych

2. Zasadniczo wszystkie definiowane przez nas zmienne czy funkcje są elementem słownika związanego z danym kontekstem. Do słowników tych można uzyskać dostęp poprzez funkcje `globals()` (słownik zawierający elementy zdeklarowane w kontekście globalnym) i `locals()` (słownik zawierający elementy zadeklarowane w kontekście lokalnym).

- to Python decyduje o tym kiedy zwalniać / ponownie użyć pamięć pozostałą po obiektach na które nie wskazuje już żadna nazwa

2.5.2 Kopiowanie obiektów

Python w momencie przypisania wartości jednej zmiennej do innej nie tworzy kopii obiektu na który wskazuje zmienna, zamiast tego przypisuje referencję do istniejącego obiektu. Jest to szczególnie zauważalne w obiektach, które mogą być wewnętrznie modyfikowalne (takich jak listy czy słowniki)[3]:

```
a = [1, 2, 3]
b = a
print(a, b, "\n", hex(id(a)), hex(id(b)))
a[1] = 0
print(a, b, "\n", hex(id(a)), hex(id(b)))
del a
print(b,    "\n", hex(id(b)))
```

```
[1, 2, 3] [1, 2, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3] [1, 0, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3]
 0x7f50d76b2bc8
```

Jak widać `a` i `b` posiadają taki sam identyfikator obiektu zwracany przez funkcję `id`, modyfikacja `a[1]` wpłynęła na zawartość `b`, natomiast usunięcie `a` nie ma wpływu na `b` (usunęliśmy tylko jedną z dwóch referencji na wspólny obiekt). Jeżeli chcemy uzyskać kopię listy lub słownika musimy skorzystać z metody `copy()` odpowiedniego obiektu:

```
a = [1, 2, 3]
b = a.copy()
b[1] = "X"
print(a, b, "\n", hex(id(a)), hex(id(b)))
```

```
[1, 2, 3] [1, 'X', 3]
 0x7f50d76b2bc8 0x7f50d57a7088
```

Zauważ że tak utworzone `b` ma inny identyfikator obiektu niż `a`. Należy mieć także na uwadze że nawet argumenty funkcji przekazywane są jako referencje na obiekty a nie kopie obiektów, natomiast dopiero operacja przypisania nowej wartości do zmiennej związanej z argumentem powoduje że zaczyna ona wskazywać na nowo utworzony (w wyniku wyrażenia po prawej stronie znaku równości) obiekt.

2.5.3 Dla jeszcze bardziej dociekliwych

Osobom jeszcze bardziej dociekliwym w temacie wnętrzności Pythona możemy polecić lekturę artykułu omawiającego te zagadnienia http://www.rwdev.eu/articles/objectthinking oraz samodzielne eksperymenty.

3 Obsługa błędów

Wcześniej spotkaliśmy się już z komunikatem błędu. Błędy mogą wynikać z błędów składniowych w programie ale również nie przewidzianych zdarzeń w trakcie jego pracy. Warto mieć na uwadze iż wszystkie błędy w Pythonie mają postać wyjątków które mogą zostać obsłużone blokiem `try`/`except`.

```
try:
  a = 5 / 0
except ZeroDivisionError:
  print("dzielenie przez zero")
except:
  print("inny błąd")
```

3. Zauważ że jedyną możliwością modyfikacji liczby czy napisu jest przypisanie wartości wyrażenia do zmiennej, a dla list czy słowników możemy je modyfikować bez operacji przypisania całej listy czy słownika do nowej czy tej samej zmiennej.

Przy obsłudze błędów może przydać się instrukcja pusta **pass**, która w tym przypadku pozwala na zignorowanie obsługi danego błędu.

```
try:
  slownik["a"] += 1
except:
  pass
```

Powyższy kod zwiększy wartość związaną z kluczem "a" w słowniku slownik, jednak gdy napotka błąd (np. słownik nie zawiera klucza "a") zignoruje go.

Możemy także generować wyjątki z naszego kodu, służy do tego instrukcja **raise**, której należy przekazać obiektem dziedziczącym po **BaseException** np:

```
raise BaseException("jakiś błąd")
```

4 Zadania

Zadanie 4.0.1

Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy wspak. Np. dla listy ['Ala', 'ma', 'kota'] funkcja powinna wypisać: alA am atok

Zadanie 4.0.2

Napisz funkcję wyiksuj(napis), która wypisze dany napis, zastępując każdą małą literę polskiego alfabetu małą literą x i każdą wielką literę polskiego alfabetu wielką literą X, natomiast resztę znaków pozostawi bez zmian. Np. dla napisu 'Python 3.6.1 (default, Dec 2015, 13:05:11)' program powinien wypisać: Xxxxxx 3.6.1 (xxxxxxx, Xxx 2015, 13:05:11)

Zadanie 4.0.3

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest słowem (tzn. nie zawiera spacji).

Zadanie 4.0.4

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest liczbą (tzn. jest złożony z cyfr i kropki, a na początku może wystąpić + albo -).

Zadanie 4.0.5

Napisz program dekodujący napis kodowany w UTF8 zakodowany przy pomocy base64 mający postać: b'UHl0aG9uIGplc3QgZmFqbnkg8J+Yjg==\n'.
Wskazówka: dane wejściowe funkcji decode() muszą być typu "bytes", można to uzyskać poprzedzając napis prefiksem b, tak jak powyżej.

Zadanie 4.0.6

Zapoznaj się z dokumentacją klasy odpowiedzialnej za napisy (`str`), zwróć szczególną uwagę na metody `split`, `find`, `replace`. Korzystając z metod klasy `str` napisz funkcję `parse` która dla napisu będącego jej argumentem wykona zamianę wszystkich ciągów "XY" na spację oraz dokona rozbicia napisu złożonego z pól rozdzielanych dwukropkiem na listę napisów odpowiadających poszczególnym polom. Funkcja powinna działać w następujący sposób:

```
 > l = parse("Ala:maXYkota:i inne:zwierzeta")
 > print(l)
['Ala', 'ma kota', 'i inne', 'zwierzeta']
```

Zadanie 4.0.7

Napisz funkcję `zlicz` która dla podanej listy policzy powtórzenia jej elementów. Przykład użycia:

```
 > zlicz(["AX", "B", "AX"])
AX wystepuje 2 razy
B wystepuje 1 razy
```

Wskazówka: Użyj słownika, w którym element będzie stanowił klucz, a krotność jego wystąpień wartość. Możesz użyć metody `get()` do pobierania wartości z słownika, jeżeli w nim jest lub wartości domyślnej w przeciwnym wypadku - szczegóły zobacz w dokumentacji

5 Zadania dodatkowe

Zadanie 5.0.1

Napisz funkcję, która dla danej listy słów wypisze w kolejnych wierszach ich skróty w postaci `<pierwsza litera>-<ostatnia litera> (<dlugosc slowa>)`.
Np. dla listy `['Interdyscyplinarne', 'Centrum', 'Modelowania']` powinna wypisać:

```
I-e (18)
C-m (7)
M-a (11)
```

Wskazówka: wynik funkcji `len()` mierzącej długość napisu jest liczbą. Do rozwiązania tego zadania może Ci się przydać konwersja tej liczby na napis (aby dało się ją skleić z innymi napisami), z użyciem funkcji `str()`

Zadanie 5.0.2

Napisz funkcję `toStr(liczba, podstawa)`, która konwertuje podaną liczbę do reprezentacji napisowej w systemie o podanej podstawie.
Wskazówka: do testowania poprawności działania możesz użyć funkcji `int(napis, podstawa)`, możemy przyjąć że podstawa jest mniejsza od 37 tak aby starczyło liter alfabetu łacińskiego.

Zadanie 5.0.3

Jak wiemy język złożony ze słów postaci `aa..aabb..bbaa..bb` (gdzie ilość liter a przed ciągiem liter b jest równa ilości liter a po tym ciągu) nie jest regularny. Jednak programistyczne wyrażenia regularne są rozszerzone w stosunku co do tych spotykanych w matematyce i umożliwiają opis takiego języka. Napisz funkcję, korzystającą z dopasowywania wyrażeń regularnych, która będzie sprawdzała czy podane słowo należy do tego języka.

6 Praca domowa

6.1 Instrukcja wysyłania rozwiązań

Rozwiązania zadań domowych należy przesłać na adres licealisci.pracownia@icm.edu.pl wpisując jako temat wiadomości `g2.x PD2`, gdzie `x` to numer grupy, np. `g2.1 PD2` dla grupy nr. 1, itd. Zadania domowe są nie obowiązkowe, jednak zachęcamy do ich robienia i wysyłania rozwiązań (nawet niekompletnych). Termin nadsyłania zdań domowych to 2020-03-03 godz. 16^{00}.

Na ten adres można także nadsyłać ewentualne pytania do zadań (zarówno domowych jak i innych zamieszczonych w skrypcie), w tym wypadku także prosimy o umieszczenie w temacie wiadomości `g2.x`, gdzie `x` to numer grupy.

6.2 Zadania domowe

Zadanie 6.2.1 — 1 pkt

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis kończy się xyz.

Zadanie 6.2.2 — 2 pkt

Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy powtarzając każdą małą literę dwukrotnie. Np. dla `['Ala', 'ma', 'kota', 'i PSA']` funkcja powinna wypisać:

```
Allaa
mmaa
kkoottaa
ii PSA
```

Zadanie 6.2.3 — 3 pkt

Napisz program, który wypisze na ekranie *trójkąt z iksów*, taki jak poniżej:

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
```

Stosując zamiast co najmniej jednej pętli rekurencję. *Wskazówka: Funkcja rekurencyjna to funkcja, która wywołuje samą siebie (typowo ze zmodyfikowanymi argumentami), dopóki zachodzi jakiś ustalony warunek (typowo zależny od argumentów).*