Akademia Górniczo-Hutnicza

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

WSTĘP DO INFORMATYKI

Adrian Horzyk

python

WPROWADZENIE

Python (https://www.python.org/) jest językiem programowania ogólnego przeznaczenia typu *open source*, zoptymalizowany pod kątem jakości, wydajność, przenośności i integracji. Jest on obecnie używany przez miliony programistów na całym świecie.

Python jest niezwykle prostym **językiem zorientowanym obiektowo (OOP – *Open Oriented Programming*)**, posiadającym czytelną składnię, łatwy w utrzymaniu i integracji z komponentami języka C, posiadający bogaty zbiór interfejsów, bibliotek i narzędzi programistycznych.

Twórcą języka Python jest **Guido van Rossum.**

Python jest językiem interpretowanym, co daje większą łatwość modyfikacji gotowego programu, lecz obniża efektywność działania w stosunku do języków kompilowanych, takich jak C.

Program źródłowy napisany w języku Python (podobnie jak w Java) może być najpierw **kompilowany do postaci pośredniej (byte-code**), która następnie wykonywana jest przez **Wirtualną Maszynę Pythona (PVM)** na konkretnej platformie obliczeniowej.

ŚRODOWISKO

Python Spyder (https://pythonhosted.org/spyder/) jest środowiskiem developerskim, które posiada wersję dla Windows, Linux i IOSa.

Spyder (Python 2.7)

File Edit Search Source Run Debug Consoles Tools View Help

C:\Python27\Scripts

Editor - C:\Users\Adrian\.spyder2\temp.py

temp.py

```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7
8
```

print "Hello World!"

spyder

Variable explorer

Name	Type	Size	Value

Object inspector | Variable explorer | File explorer

IPython console

Console 1/A

```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32
bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra
details.
%guiref   -> A brief reference about the graphical user
interface.

In [1]:
```

Console | History log | IPython console

Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 1 Column: 1 Memory: 47 %

python STAŁE, LICZBY I ŁAŃCUCHY

- Liczby: **5, 3.24, 9.86e-3, (-5 + 2j)**
- Łańcuchy: **'Łańcuch'**, **"Łańcuch"** – w apostrofach lub cudzysłowiach
- **" " "** Za pomocą potrójnych cudzysłowów można oznaczać łańcuchy wielolinijkowe **" " "**
- W łańcuchach można używać znaków specjalnych, np. znaku nowej linii: \n " Pierwszy wiersz.\nDrugi wiersz. "
- Jeśli łańcuch poprzedzimy znakiem **r** lub **R**, wtedy nie są uwzględniane znaki specjalne, a tekst jest traktowany dosłownie: **R" …\n…"** – tzw. łańcuch surowy.
- Chcąc poprawnie wyświetlać polskie znaki diakrytyczne najlepiej posłużyć się unikodem sygnalizując to znakiem **u** lub **U**: **U"Tekst zapisaliśmy w Unikodzie"**
- Python automatycznie łączy łańcuchy obok siebie: **'Adrian' 'Horzyk'** zostanie automatycznie przekonwertowany przez interpreter Pythona na **'Adrian Horzyk'**
- Łańcuchy w Pythonie są niezmienne, lecz istnieją metody operowania na nich, tworząc nowe z części starych.

ZMIENNE

Zmienne umożliwiają przechowywanie w pamięci komputera dane, które mogą zmieniać się w czasie, np. w wyniku obliczeń.

Zmienne posiadają swoje nazwy, które je identyfikują:

- Pierwszym znakiem identyfikatora musi być mała lub duża litera alfabetu (łacińskiego) albo podkreślnik (_), lecz takie zmienne mają specjalne znaczenie.
- Pozostałe znaki mogą zawierać małe lub duże litery alfabetu łacińskiego, podkreślniki oraz cyfry (0–9).
- Wielkość znaków w identyfikatorze jest ważna, stąd **nazwaZmiennej** i **nazwazmiennej** to dwie inne zmienne.
- Przykłady poprawnych identyfikatorów to: i, __moja_zmienna, nazwa_123, a1.
- Nazwy zmiennych nie mogą rozpoczynać się od cyfry, zawierać spacje ani łączniki: 3A, nazwa ze spacjami, nazwa-z-lacznikiem.

OPERATORY

Operatory służą do wykonywania operacji matematycznych, logicznych i symbolicznych:

- Dodawanie: + — Dodawanie do dotychczasowej wartości: a += 2
- Odejmowanie: - — Odejmowanie od dotychczasowej wartości: a -= 2
- Mnożenie: * — Mnożenie przez dotychczasową wartość: a *= 2
- Druga potęga: **
- Dzielenie: / — Dzielenie przez dotychczasową wartość: a /=
- Dzielenie całkowite: // — Dzielenie, po którym część ułamkowa jest obcinana: 9 // 2 = 4
- Reszta z dzielenia: %
- Dodawanie zbiorów: +
- Multiplikacja zbiorów: *
- Suma zbiorów: |
- Iloczyn zbiorów: &
- Odejmowanie zbiorów: -
- Negacja: not — == równość, != nierówność,
- Alternatywa: or — >, <, >=, <= znaki nierówności liczbowej lub leksykograficznej
- Koniunkcja: and

Operatory podsumowanie: https://www.tutorialspoint.com/python/pdf/python_basic_operators.pdf

INSTRUKCJA WARUNKOWA

Instrukcja warunkowa:

```
if WARUNEK1:            # Jeśli jest spełniony wykonaj blok instrukcji
    Instrukcja1
    Instrukcja2
elif WARUNEK2:          # Jeśli nie to zbadaj kolejny warunek
    Instrukcja3
    Instrukcja4
else:                   # Jeśli nie jest spełniony wykonaj blok instrukcji
    Instrukcja5
    Instrukcja6
Instrukcja7             # Instrukcja następna po instrukcji warunkowej
```

Bloki instrukcji w Pythonie wyróżniane są za pomocą odpowiedniego wcięcia (tabulatora). Nie stosuj się tutaj żadnych znaków specjalnych, jak w innych językach programowania!

Części **elif** oraz **else** są opcjonalne.

Każda część instrukcji warunkowej kończy się dwukropkiem!

ZAGNIEŻDŻANIE i BLOKI

Instrukcje warunkowe można zagnieżdżać, co zaznaczone jest poprzez odpowiednio duże wcięcia (ilość tabulatorów) wyznaczające zagnieżdżone bloki instrukcji:

```
if WARUNEK1:                    # Jeśli jest spełniony wykonaj blok instrukcji
    Instrukcja1
    if WARUNEK2:                # Zagnieżdżona instrukcja warunkowa
        Instrukcja2
    else:                       # else zagnieżdżonej instrukcji warunkowej
        Instrukcja3
else:                           # else nadrzędnej instrukcji warunkowej
    Instrukcja4
```

Zakończenie bloku instrukcji sygnalizowane jest poprzez zmniejszenie wcięcia.

INSTRUKCJA PĘTLI WHILE

Instrukcja pętli **while** pozwala na kilkukrotne wywołanie bloku poleceń tak długo, jak długo warunek będzie prawdziwy:

```
while WARUNEK:                 # Jeśli jest spełniony wykonaj blok instrukcji
        Instrukcja1
        Instrukcja2
else:                          # Element else jest opcjonalny
        Instrukcja3            # Instrukcja3 wykonana po zakończeniu pętli
Instrukcja4                    # Instrukcja następna po instrukcji pętli
```

Instrukcja3 po **else** jest zawsze wykonywana za wyjątkiem sytuacji, gdy pętla jest przerwana poleceniem **break**.

Każda część instrukcji warunkowej kończy się dwukropkiem!

python INSTRUKCJA PĘTLI FOR … IN

Instrukcja pętli **for … in** służy do iterowania po dowolnej sekwencji obiektów, gdzie sekwencja to uporządkowany zbiór elementów:

```
for i in range (1,5):               # i przyjmuje kolejne wartości od 1 do 4
        Instrukcja1
        Instrukcja2
else:                               # Element else jest opcjonalny
        Instrukcja3                 # Instrukcja3 wykonana po zakończeniu pętli
Instrukcja4                         # Instrukcja następna po instrukcji pętli
```

Instrukcja3 po **else** jest zawsze wykonywana za wyjątkiem sytuacji, gdy pętla jest przerwana poleceniem **break**.

Istnieje również możliwość przerwania aktualnie wykonywanego cyklu pętli (czyli pominięcie pozostałych instrukcji pętli) i kontynuować jej wykonywanie dla kolejnego elementu iterowanej sekwencji poleceniem **continue**.

Każda część instrukcji warunkowej kończy się dwukropkiem!

Funkcja **range (x,y)** zwraca kolejne liczby rozpoczynając od x do liczby poprzedzającej y.

for i **in range (x,y) w Pythonie** odpowiada instrukcji **for (int i=x; i<y; i++) w C++**

PRZERYWANIE PĘTLI i CYKLU

Przerywanie pętli poleceniem **break**
oraz przerywanie aktualnego cyklu pętli poleceniem **continue**:

```
for i in range (1,10):                    # i przyjmuje kolejne wartości od 1 do 9
        s = raw_input('Wpisz coś: ')
        if len(s) > 10:                   # Funkcja len(s) zwraca długość łańcucha s.
                print "Tekst jest za długi!"
                break                     # Przerywamy!
        elif len(s) < 3 :
                print "Tekst jest za krótki!"
                continue                  # Ignorujemy kolejne instrukcje w bloku pętli!
        print "Tekst ma prawidłową długość, tzn. 3 do 10 znaków."
else:
        print "Jeśli nie wpisałeś zbyt długiego tekstu zobaczysz ten komunikat."
print "Następna instrukcja po zakończeniu pętli."
```

Funkcja len(s) może zwracać niepoprawne wartości dla polskich znaków # diakrytycznych (‘ą’, ‘ę’, ‘ó’, ‘ł’, ‘ś’, ‘ć’, ‘ń’, ‘ź’, ‘ż’), mnożąc każdy z nich przez 2 lub 4, jeśli nie poinformujemy Pythona o stosowaniu Unicode, np. **u'Łoś'**

FUNKCJE

Funkcje umożliwiają wyciąć pewien fragment kodu w celu jego wielokrotnego użytku w różnych miejscach w programie. Takim fragmentom kodu nadajemy nazwę, która będzie służyła do wywołania tego fragmentu kodu z opcjonalną listą parametrów ujętych w nawiasach. Funkcja w Pythonie może zwracać jedną lub więcej wartości:

```
def jakCiSiePodobaWyklad(nazwa):          # Definiujemy funkcję i jej nazwę
        if nazwa == 'WDI':
                print u'Wykład jest super!'
        else:
                print u'Jeszcze nie wiem?'


nazwa = raw_input('Wpisz skrót wykładu: ')
jakCiSiePodobaWyklad(nazwa)        # Wywołanie funkcji w kodzie
```

Wykorzystujemy w funkcji zmienną lokalną nazwa, której wartości nie mają wpływu na wartości zmiennej o tej samej nazwie w programie głównym.

Zmienna lokalna o tej samej nazwie co globalna przykrywa ją chwilowo.

GENERATORY

W Pythonie można zdefiniować sobie tzw. **generatory**, które są mechanizmem leniwej ewaluacji funkcji, tzn. zamiast zwracać obciążającą pamięć listę, generator jest obiektem przechowującym stan ostatniego wywołania, mogącym wielokrotnie wchodzić i opuszczać ten sam dynamiczny zakres. Wywołanie generatora umożliwia więc zwracanie kolejnych elementów z dynamicznego zakresu zamiast całej listy, co daje wygodny mechanizm do iterowania po kolejnych elementach dynamicznie generowanych elementów listy:

```
def kwadraty(N):  # Definiujemy generator N kwadratów liczb całkowitych
        for i in range(N):
                yield i**2
```

Generator zamiast słowa kluczowego **return** stosuje **yield** zwracający kolejny wygenerowany element ciągu / listy.

Można to zapisać również w postaci:

```
kwadraty = (i**2 for i in range(N))
```

ZMIENNE LOKALNE I GLOBALNE

Zmienne różnią się zasięgiem swojego działania jak również okresem istnienia.

Można poinformować interpreter Pythona, iż ma operować na zmiennej globalnej, za pomocą słowa kluczowego global:

```
def jakCiSiePodobaWyklad():          # Do funkcji nie przekazujemy parametry
    global nazwa
    nazwa = raw_input('Wpisz skrót wykładu: ')  # zmieniamy wartość globalnie
    if nazwa == 'WDI':
        print u'Wykład jest super!'
    else:
        print u'Jeszcze nie wiem?'


nazwa = 'SI'                         # nazwa jest tutaj zmienną globalną
jakCiSiePodobaWyklad()               # Wywołanie funkcji w kodzie
print ' nazwa = ', nazwa             # Wypisze tekst wprowadzony wewnątrz funkcji!
```

PARAMETRY DOMYŚLNE

Do funkcji można przesyłać wiele argumentów, z których niektóre mogą być domyśle, jeśli nie podane w trakcie wywołania funkcji. Parametrom o domyślnych wartościach przypisujemy wartość domyślą przy pomocy znaku przypisania:

```
def jakCiSiePodobaWyklad(nazwa = 'WDI'):   # Wartością domyślą będzie 'WDI'
        if nazwa == 'WDI':
                print u'Wykład jest super!' * 3        # Łańcuch wypisze 3x
        else:
                print u'Jeszcze nie wiem?'


nazwa = raw_input('Wpisz skrót wykładu: ')
if len(nazwa) < 2:
        jakCiSiePodobaWyklad()                 # Wywołanie funkcji bez parametru
else:
        jakCiSiePodobaWyklad(nazwa)            # Wywołanie funkcji z parametrem
```

Wartości parametrów domyślnych muszą być niezmienne (stałymi), czyli nie mogą być zmienną ani wyznaczane w wyniku obliczeń czy innych operacji.

PARAMETRY DOMYŚLNE

Jeśli funkcja posiada wiele parametrów, w tym parametry domyślne, a ty nie chcesz podawać wszystkich, wtedy podajesz ich nazwę jawnie w celu wskazania, które parametry przekazujesz:

```
def suma(a = 'WDI', b = 'jest', c):     # Kilka wartości domyślnych
        print 'Wykład z' a b c '!'      # Łączenie łańcuchów


suma (c = 'ciekawy')                    # Jawne podanie nazwy parametru


suma (c = 'intrygujący', a = 'SI')              # Jawne podanie nazwy parametru
```

Wartości parametrów domyślnych muszą być niezmienne (stałymi), czyli nie mogą być zmienną ani wyznaczane w wyniku obliczeń czy innych operacji.

Można nawet zmieniać kolejność parametrów, jeśli jawnie podajemy, któremu jaką wartość przypisujemy.

ZWRACANIE WARTOŚCI

Funkcje mogą zwracać wartości. Celem wskazania tych wartości lub zmiennych stosujemy słowo kluczowe **return**:

```
def maksimum(x, y):                # Definiujemy funkcję i jej nazwę
        if x > y:
                return x
        else:
                return y

print 'Maksimum z liczb', x, y, 'wynosi', maksimum(x,y)
```

Możemy zwrócić nawet więcej niż jedną wartość, np. krotkę zawierającą wiele wartości.

Każda funkcja domyślnie zwraca wartość **None**, chyba że wykorzystamy **return**.

None to specjalny wartość w Pythonie, która oznacza po prostu nic / brak wartości.

MODUŁY

Funkcje można agregować i tworzyć moduły (biblioteki funkcji), które można zaimportować do wielu różnych programów przy pomocy instrukcji **import**.

Istnieje również standardowa biblioteka Pythona, którą importujemy następująco:

import sys

Do funkcji i zmiennych wewnątrz modułu odwołujemy się przy pomocy kropki:

sys.path

sys.argv # to lista ciągów znaków

Wskazując wyraźnie przed kropką z jakiego modułu ta funkcja lub zmienna pochodzi.

Możemy równocześnie zaimportować wiele funkcji z różnych modułów.

Chcąc uzyskać informacje o katalogu, w którym jesteś posłuż się:

import os

print os.getcwd()

Moduły można kompilować w celu przyspieszenia operacji na nich, wtedy posiadają rozszerzenie .pyc.

IMPORTOWANIE

Można wybiórczo importować funkcje z wybranych modułów instrukcją
from … import …

np.:

from sys **import** argv

lub

Zaimportować wszystkie funkcje z wybranego modułu:

from sys **import** *

Należy jednak unikać importowania wszystkiego z każdego modułu, gdyż powoduje to zaśmiecenie przestrzeni nazw a ponadto może dojść do konfliktu nazw pomiędzy różnymi modułami.

Importujemy tylko najczęściej stosowane funkcje w celu ominięcia konieczności stosowania przedrostków wskazujących na nazwy modułów.

TYPY ZMIENNYCH

Python pozwala używać zmiennych do przechowywania wartości dowolnego typu.

Utworzenie zmiennej polega na nadaniu jej wartości początkowej:

a = 4

Usunięcie zmiennej wykonujemy instrukcją del:

del a

Zmienne nie mogą posiadać **nazw zastrzeżonych** dla Pythona:

and assert break class continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return try while yield

KONWERSJE

Konwersja liczb na łańcuchy – dokonywana jest przez odwrócony apostrof:

a = 2

b = 8

a+b

da w efekcie

10

zaś

`a` + `b`

da w efekcie

'28'

Konwersja łańcuchów na liczby – dokonywana jest z wykorzystaniem funkcji:

x = '2'

y = '8'

x+y

da w efekcie

'28'

zaś

int(a) + int(b)

da w efekcie

10

Funkcje konwersji łańcuchów na liczby:

int(x)	- na liczby całkowite
long(x)	- na duże liczby całkowite
float(x)	- na liczby wymierne
complex(x)	- na liczby zespolone

WPROWADZANIE DANYCH

Podstawą każdego programu są **dane**, które są przetwarzane w trakcie działania programu. Żeby więc wprowadzić dane z klawiatury stosujemy funkcję:

x = **raw_input** ("Podaj wartość ")

która przypisuje łańcuch tekstu do podanej zmiennej.

Chcąc więc wykonywać działania arytmetyczne na tak wprowadzonych danych należy dokonać konwersji korzystając z odpowiedniej funkcji:

int(x) - na liczby całkowite

long(x) - na duże liczby całkowite

float(x) - na liczby wymierne

complex(x) - na liczby zespolone

Drugą podstawową operacją jest wypisywanie wyników działania programu:

print "To jest wynik", x

OPERATORY LOGICZNE

Operatory logiczne:

not – negacja

and – iloczyn logiczny

or – suma logiczna

Priorytety i kolejność wykonywania operatorów:

1. Operatory porównania <, >, <=, >=, ==, !=, <>
2. Operator negacji (not)
3. Operator iloczynu logicznego (and)
4. Operator sumy logicznej (or)

W przypadku woli zmiany kolejności wykonywania operatorów niezbędne jest zastosowanie nawiasów okrągłych.

KODY ASCII ZNAKÓW

Łańcuchy składające się **z pojedynczych znaków** można traktować jako **znaki** i przypisać im kod ASCII funkcją **ord(c):**

ord('A') zwróci 65

ord('1') zwróci 49

Odwrotną zamianę kodów ASCII na odpowiadające im znaki wykonamy funkcją **chr(I)**:

chr(65) zwróci 'A'

chr(49) zwróci '1'

chr(10) zwróci '\n'

ŁAŃCUCHY

Łańcuchy można mnożyć i dodawać:

h = " Huraaa!"   # Uwaga: 1. znak jest spacją!

" Znowu wykład z WDI! " + 3*h

da w efekcie:

Znowu wykład z WDI! Huraaa! Huraaa! Huraaa!

Długość łańcucha wyznaczamy funkcją **len(**h**)**, otrzymując w tym przypadku 8.

Poszczególne znaki w łańcuchu można odczytywać od początku lub od końca.
Od początku są indeksowane liczbami: 0, 1, 2, 3…, a od końca: -1, -2, -3 itd.

h[2] zwróci 'u'

h[-5] zwróci 'r'

W podobny sposób możemy pobrać **część łańcucha** od przodu lub od tyłu podając interesujący nas przedział indeksów stosując **dwukropek**:

h[:3] zwróci ' Hu'

h[3:] zwróci 'raaa!'

h[3:5] zwróci 'ra'

h[1:7:2] zwróci co drugi znak z podanego przedziału 'Hra'

h[-6:-2] zwróci 'uraa' a więc można też stosować do wycinania ujemne indeksy

Łańcuchy można mnożyć i dodawać:

s = 'to jest fajna UCZELNIA!'

s.**capitalize()** zamienia pierwszą literę w zdaniu na dużą a pozostałe zmienia na małe:

'To jest fajna uczelnia!'

Zamianę na małe lub duże znaki osiągniemy dzięki metodom **lower()** i **upper()**:

s = 'to jest fajna UCZELNIA!'

s.**lower()** zwróci: 'to jest fajna uczelnia!'

s.**upper()** zwróci: 'TO JEST FAJNA UCZELNIA!'

Metoda **swapcase()** odwraca wielkość liter w napisie:

s.**swapcase()** zwróci: 'TO JEST FAJNA uczelnia!'

Metoda **title()** zmienia wielkość liter jak w tytule:

s.**title()** zwróci: 'To Jest Fajna Uczelnia!'

Metoda **replace** zamienia wszystkie wystąpienia określonego ciągu znaków na inny:

s.**replace(**'UCZELNIA!', 'dziewczyna'**)** zwróci: 'to jest fajna dziewczyna'

Metoda **join()** połączy poszczególne elementy listy wskazanym znakiem (tutaj spacją):

lista = ['jabłko', 'gruszka', 'śliwka', 'morela', 'mango', 'ananas', 'brzoskwinia']

' '.**join(**lista**)**

'jab\xc5\x82ko gruszka \xc5\x9bliwka morela mango ananas brzoskwinia,

Metody wyrównujące:

s.**rjust()**	wyrównuje łańcuch do prawej strony dodając spacje od lewej
s.**center**(64)	służy do wyśrodkowania tekstu w polu o zadanej długości
s.**center(**64, '*'**)**	dodatkowo wypełnia miejsca puste wskazanym znakiem '*'

'********************to jest fajna UCZELNIA!*********************'

Metody wyszukujące i zliczające:

s.**find(**'a'**)**	wskazuje indeks pierwszego wystąpienia wskazanego ciągu znaków
s.**rfind(**'n'**)**	znajduje ostatnie wystąpienie określonego ciągu znaków: 11 albo -1, jeśli takie nie odnajdzie w całym ciągu.
s.**isdigit()**	sprawdza, czy łańcuch zawiera tylko cyfry
s.**count(**'a'**)**	zwraca ilość wystąpień łańcucha (tutaj 'a') w łańcuchu s

DZIELENIE ŁAŃCUCHÓW

Metoda **split()** tworzy listę wyrazów z elementów łańcucha:

s = 'to jest fajna UCZELNIA!'

s.split() tworzy listę wyrazów z łańcucha: ['to', 'jest', 'fajna', 'UCZELNIA!']

ss = '48-12-5678943'

data = '18.10.2016'

ss.split('-') możemy wskazać znak rozdzielający: ['48', '12', '5678943']

data.split('.') możemy wskazać znak rozdzielający: ['18', '10', '2016']

Metoda **splitlines()** dzieli tekst na linie :

((s+'\n') * 6).**splitlines()**

['to jest fajna UCZELNIA!', 'to jest fajna UCZELNIA!', 'to jest fajna UCZELNIA!', 'to jest fajna UCZELNIA!', 'to jest fajna UCZELNIA!', 'to jest fajna UCZELNIA!']

python TYPY ZMIENNE I NIEZMIENNE

W Pythonie typy danych dzielimy na:

zmienne (mutable) – mogą zmieniać swoje wartości

niezmienne (immutable) – nie mogą zmieniać swojej wartości

Łańcuchy w Pythonie należą do typów niezmiennych!

Nie można więc podmieniać znaków w łańcuchu, ani do nich nic dodawać lub obcinać, lecz tylko utworzyć nowy łańcuch zawierający podmienione, dodane lub obcięte znaki, np.:

t1 = 'tak'

t1 = t1 + 'tyka'   tworzy nowy łańcuch t1 na podstawie starego łańcucha t1

zaś zapis t1 += 'tyka' jest błędny

t1 = t1[:3]   zwróci nowy łańcuch t1 o obciętej zawartości starego t1

STRUKTURY DANYCH

W celu umożliwienia reprezentacji danych powiązanych ze sobą różnymi relacjami potrzebne są struktury danych pozwalające odwzorować te zależności.

Do podstawowych struktur danych w Pythonie należą:

Lista – to struktura liniowa zawierająca uporządkowany zestaw obiektów. Lista jest **zmiennym typem danych** umożliwiającym dodawanie i usuwanie elementów z listy. Elementy listy zapisujemy w nawiasach kwadratowych.

lista = ['jabłko', 'gruszka', 'śliwka', 'morela', 'mango', 'ananas', 'brzoskwinia']

Krotka – to **typ niezmienny**, a więc nie można jej modyfikować po jej utworzeniu. Elementy krotki zapisujemy w nawiasach okrągłych. Jeśli krotka jest jednoelementowa, obligatoryjnie stawiamy przecinek na końcu, np. pora = ('noc',)

krotka = ('jabłko', 'gruszka', 'śliwka', 'morela', 'mango', 'ananas', 'brzoskwinia')

Słownik – to **tablica asocjacyjna** przechowująca klucze i przypisane im wartości, przy czym klucze muszą być unikalne. Zbiór takich par ujęty jest w nawiasy wężykowe:

slownik = {klucz1 : wartość1, klucz2 : wartość2, …, kluczN : wartośćN}

Słowniki nie sortują automatycznie danych, więc jeśli dane mają być posortowane, trzeba to zrobić zewnętrznie.

Lista, krotka i słownik – to **sekwencje**, których elementy są ponumerowane, więc można się do nich odwoływać po indeksie lub po nich iterować pętlą for … in …:

STRUKTURY DANYCH

Zbiór – to nieuporządkowany zestaw prostych obiektów, który umożliwia wykonywanie operacji na zbiorach

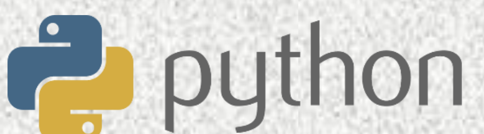
Liga_mistrzow = **set** (['Polska', 'Brazylia', 'Niemcy'])

oraz sprawdzanie występowania elementu w nim:

'Polska' in Liga_mistrzow >>> true

'Węgry' not in Liga_mistrzow >>> true

Odniesienie – to wskaźnik do miejsca w pamięci komputera, w którym znajdują się określone dane. Tworzenie odniesień to wiązanie (bindowanie) nazwy z obiektem.



LISTY

Lista – to typ zmienny, więc można podmieniać jej elementy, dodawać i usuwać.

Lista może też być pusta:

lista = []

Listy mogą zawierać elementy różnych typów:

x = 34; y = 12; z = 18

lista = [1, 3.5, x, 'Ania']

Można tworzyć listę list lub lista może zawierać inną listę jako swój element:

listalist = [[1, 2, 3, 4], [x, y, z], ['wtorek', 'czwartek', 'sobota']]

Jej długość obliczymy przy pomocy funkcji **len(lista)**:

len(lista) zwróci 4

len(listalist) zwróci 3, gdyż zawiera 3 elementy, które są listami

Dostęp do elementów listy uzyskujemy w podobny sposób jak dla łańcuchów:

lista[2] zwróci 34

listalist[1][2] zwróci 18

lista[-1] zwróci 'Ania'

lista[1:2] zwróci [3.5, 34]

lista[2:] zwróci [34, 'Ania']

LISTY

LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]

LiczbyNaturalne[1::3] zwróci co trzeci element list począwszy od drugiego do końca, a więc: [2, 5, 8]

Listy można **powielać**:

LiczbyNaturalne *= 2

Spowoduje **zwielokrotnienie**:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Elementy można **podmieniać**:

LiczbyNaturalne[9:18]=[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Spowoduje **podmianę** wskazanych elementów listy:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Można też wykonywać operacje na poszczególnych jej elementach

```
for i in range (0,19):
        LiczbyNaturalne[i]*=2
```

Spowoduje przemnożenie każdego z jej elementów przez 2:

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]

del LiczbyNaturalne[9:17] skróci listę do [2, 4, 6, 8, 10, 12, 14, 16, 18, 36, 38]

OPERACJE NA LISTACH

Listy można porównywać:

- Listy są równie, jeśli wszystkie elementy obu list są równe.
- Listy są porównywane w porządku leksykograficznym, najpierw pierwsze, potem drugie i kolejne elementy, a dzięki temu można wyznaczyć, która lista jest większa lub mniejsza.
- Nie decyduje tutaj długość listy.
- Element nieistniejący jest zawsze mniejszy od każdego innego elementu

Sprawdzanie, czy lista **zawiera lub nie zawiera pewien element**:

LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]

1 **in** lista LiczbyNaturalne zwróci true

0 **not in** lista LiczbyNaturalne zwróci true

Wszystkie sekwencje zmienne w Pythonie wskazują na miejsce w pamięci, w którym te zmienne się znajdują (**wskaźniki**), więc przypisanie:

NowaLista = LiczbyNaturalne

przepisze jedynie wskaźnik, a nie zrobi kopii elementów listy LiczbyNaturalne.

OPERACJE NA LISTACH

Metoda **range(x,y)** tworzy listę elementów:

Lista = range(1,10) daje: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Metoda **append()** dołącza do listy pojedynczy element na jej końcu:

Lista.append(10) daje: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Metoda **extend()** dołącza do listy inną listę na jej końcu:

Lista.extend([10, 11]) daje: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11]

Metoda **insert(i, w)** wstawia do listy wartość w na pozycję i:

Lista.insert(5, 10) daje: [1, 2, 3, 4, 5, 10, 6, 7, 8, 9, 10, 10, 11]

Metoda **pop(i)** zwraca wartość z i-tej pozycji listy i równocześnie usuwa ją z listy:

Lista.pop(7) daje: 7

Metoda **remove(w)** usuwa z listy pierwszą wartość w:

Lista.remove(10) daje: [1, 2, 3, 4, 5, 6, 8, 9, 10, 10, 11]

OPERACJE NA LISTACH

Metoda **count(w)** liczy, ile razy występuje w liście wartość w:

Lista.count(10) daje: 2

Metoda **index(w)** znajduje indeks pierwszego wystąpienia wartości w w liście:

Lista.index(10) daje: 8

Można też ograniczyć przeszukiwanie do wybranej części listy **index(w, od, do)**:

Lista.index(10, 9, 10) daje: 9

Metoda **reverse()** odwraca kolejność elementów listy:

Lista.**reverse()** daje: [11, 10, 10, 9, 8, 6, 5, 4, 3, 2, 1]

Metoda **sort()** porządkuje elementy listy w kolejności rosnącej:

Lista.**sort()** daje: [1, 2, 3, 4, 5, 6, 8, 9, 10, 10, 11]

LICZBY LOSOWE

Po zaimportowaniu modułu random:

import random

można generować liczby pseudolosowe, uruchamiając generator:

random.seed()

Metoda **randint(od, do)** generuje liczbę pseudolosową z podanego zakresu liczb:

random.randint(1, 10) daje np.: 9

random.randint(1, 10) daje np.: 3

W celu uniknięcia zapisu z kropką możemy zaimportować do programu najczęściej wykorzystywane funkcje, np.:

from random import randint

a wtedy możemy już wywoływać tą funkcję bez podawania nazwy modułu:

randint(1, 10) daje np.: 7

Można też zażądać zaimportowania wszystkich nazw z modułu, np.:

from random import *

Jeśli w różnych modułach importowane nazwy się powtarzają, nie da się tak zrobić!

WARTOŚCI LOSOWE

Losowy element możemy też wybrać z listy:

Lista = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

Wtedy metoda **random.choice(Lista)** zwraca losowy element tej listy, np.: 'E'

Ta sama metoda może też wybrać losowy element z sekwencji:

Sekwencja = 'Ala ma kota!'

random.choice(Sekwencja) zwróci np.: 'a'

Metoda **random.shuffle()** zwraca losową permutację wskazanej sekwencji:

random.shuffle(Lista) zwróci np.: ['F', 'E', 'D', 'B', 'C', 'A', 'H', 'G']

Metoda **random.uniform(a,b)** zwraca zmiennopozycyjną losową liczbę wymierną ze wskazanego zakresu [a, b):

random.uniform(1, 100) zwróci np.: 43.43685154212699

FUNKCJE MATEMATYCZNE

Moduł **math** zawiera definicje najczęściej używanych funkcji matematycznych:

from math import *

ceil(x)	zwraca sufit z liczby rzeczywistej x, najmniejszą nie mniejszą liczbę całkowitą
floor(x)	zwraca podłogę z liczby rzeczywistej x, największą nie mniejszą liczbę całkowitą
fabs(x)	zwraca wartość absolutną z liczby rzeczywistej x
modf(x)	zwraca krotkę zawierającą część ułamkową i część całkowitą liczby x
exp(x)	zwraca e do potęgi x
log(x)	zwraca logarytm naturalny z x
log(x, p)	zwraca logarytm przy podstawie p z liczby x
pow(x, p)	zwraca p-tą potęgę liczby x
sqrt(x)	zwraca pierwiastek z x
sin(x)	zwraca sinus z x
asin(x)	zwraca arcus sinus z x
cos(x)	zwraca cosinus z x
acos(x)	zwraca arcus cosinus z x
tan(x)	zwraca tangens z x
atan(x)	zwraca arcus tangens z x
hypot(x, y)	zwraca odległość punktu o współrzędnych (x, y) od początku układu współrzędnych (0, 0)
degrees(x)	zwraca w stopniach miarę kąta x wyrażoną w radianach
radians(x)	zwraca w radianach miarę kąta x wyrażoną w stopniach

KROTKI

Krotka – to typ niezmienny, więc nie można podmieniać jej elementów, dodawać ani ich usuwać. Pozostałe cechy krotek przypominają listy

Krotki mogą zawierać różne elementy również te o typie zmiennym:

x = 34; y = 12; z = 18

LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]

krotka = (1, 3.5, x, **'**Ania**'**, LiczbyNaturalne)

krotka może zawierać listę typu zmiennego, co oznacza, że listy zmienić nie możemy, natomiast możemy wykonywać operacje na elementach listy jak na typie zmiennym:

Dostęp do poszczególnych elementów krotki uzyskujemy podobnie jak w liście:

krotka[1] zwróci 3.5

krotka[4][5]*=2 dokonuje operacji na liście wewnątrz krotki, uzyskując w wyniku:

(1, 3.5, 34, 'Ania', [1, 2, 3, 4, 5, **12**, 7, 8, 9])

len(krotka) zwróci ilość jej elementów, czyli 5, gdyż lista jest traktowana jako jeden element krotki.

Do elementów krotki uzyskujemy dostęp podobnie jak dla listy:

krotka[4:] zwróci krotkę jednoelementową składającą się z listy:

([1, 2, 3, 4, 5, 12, 7, 8, 9]**,**) a jednoelementowe krotki kończą się zawsze **przecinkiem**

KROTKI

x = 34; y = 12; z = 18

LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]

krotka = (1, 3.5, x, 'Ania', LiczbyNaturalne)

Skrócenie krotki można wykonać tylko przez utworzenie nowej krotki na podstawie starej obciętej o pożądaną ilość elementów, np.:

krotka = krotka[:4]

krotka zwróci: (1, 3.5, x, 'Ania')

Przedłużenie krotki również wymaga stworzenie nowej przez dodanie do starej krotki nowych elementów:

krotka = krotka + (5,)

Zwróci: (1, 3.5, 34, 'Ania', 5)

W przypadku krotek dochodzi do fizycznego kopiowania elementów, więc takie operacje na dużych krotkach są bardziej czasochłonne niż na listach:

krotka2 = krotka

KONWERSJE SEKWENCJI

krotka = (1, 2, 3, 4)

lancuch = 'abcd'

list – zamienia typ sekwencyjny na listę:

lista1 = list(krotka) zwróci [1, 2, 3, 4]

lista2 = list(lancuch) zwróci ['a', 'b', 'c', 'd']

tuple – zamienia typ sekwencyjny na krotkę:

krotka = tuple(lista2) zwróci ('a', 'b', 'c', 'd')

str zamienia typ sekwencyjny na napis (uwaga nie jego elementy):

napis1 = str(lista1) zwróci '[1, 2, 3, 4]'

napis2 = str(lista2) zwróci "['a', 'b', 'c', 'd']"

napis3 = str(krotka) zwróci "('a', 'b', 'c', 'd')"

lub krócej przy pomocy **odwróconych apostrofów**:

napis4 = `lista1` zwróci '[1, 2, 3, 4]'

FORMATOWANIE

Do formatowania wykorzystywany jest operator % w połączeniu z ciągiem formatującym:

print "%s" % 10	konwertuje każdy typ danych na łańcuch/tekst
zwróci 10	
print "%c" % 33	oznacza pojedynczy znak w kodzie ASCII
zwróci !	
print "%i" % 2.8	konwertuje na liczbę całkowitą
zwróci 2	
print "%x" % 10	konwertuje na liczbę szesnastkową
zwróci a	
print "%o" % 10	konwertuje na liczbę ósemkową
zwróci 12	
print "%e" % 10	konwertuje na postać naukową
zwróci 1.000000e+01	
print "%f" % 10	konwertuje na liczbę zmiennopozycyjną
zwróci 10.000000	

FORMATOWANIE

Stałą szerokość pola do wyświetlania liczb uzyskamy dzięki:

```
for x in range(1,100,10):
        print "%4i%6i%8i" % (x,x**2,x**3)
```

uzyskując:

```
   1     1       1
  11   121    1331
  21   441    9261
  31   961   29791
  41  1681   68921
  51  2601  132651
  61  3721  226981
  71  5041  357911
  81  6561  531441
  91  8281  753571
```

Możemy też formatować ilość miejsc po przecinku liczb zmiennopozycyjnych:

```
print "Pierwiastkiem liczby %2i jest %5.3f" % (34, 34**0.5)
```

daje w wyniku: Pierwiastkiem liczby 34 jest 5.831

FORMATOWANIE

+ – wymusza wyświetlanie znaku liczby, także dla liczb dodatnich:

```
for x in range (-5,5):
        print "%+i" % x,
```

daje: -5 -4 -3 -2 -1 +0 +1 +2 +3 +4

Wyświetlenie tabliczki mnożenia osiągniemy przez:

```
for x in range(1,11):
  print # przejście do nowego wiersza
  for y in range(1,11):
    print "%3i" % (x*y),
```

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

ITEROWANIE PO …

W Pythonie możemy łatwo iterować po elementach list i krotek używając **for … in …**:

```
for a in lista1:                # interowanie po wszystkich elementach listy
    print a
for a in lista1[10:50]:         # interowanie po wybranych elementach listy od 10 do 50
    print a
for a in lista1[::2]:           # interowanie po co drugim elemencie listy
    print a
for nr, wartosc in enumerate(lista1):   # interowanie po wszystkich elementach listy
    print nr, wartosc           # z możliwością zwrócenia również indeksu elementów
for b in krotka1:
    print b
for b in krotka1[4:9:2]:        # interowanie po co drugim elemencie krotki od 4 do 9
    print b
for c in range(2,20):           # interowanie po elementach przedziału liczbowego
    print c
for c in range(2,20,3):         # interowanie po co trzecim elemencie przedziału
    print c
```

POWTÓRZENIA

W Pythonie możemy też wykorzystać pętlę **while** w przypadku, gdy nie jesteśmy w stanie określić stałej ilości powtórzeń:

```
d = [1, 2, 3, 4, 5, 6, 7]
while d:
    d = d[:len(d)-1]        # w każdym kroku zmniejszamy listę o jeden
    print d
```

otrzymując:

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

Wiele rezultatów funkcji

W Pythonie funkcje mogą zwracać **wiele rezultatów** w postaci krotki:

```
def sumypar(a, b, c):
    return a+b, b+c, a+c
```

W wyniku wywołania takiej funkcji otrzymamy krotkę:

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

Możemy też iterować pętlą po elementach takiej krotki:

```
for n in sumypar(1,2,3):   print n
```

otrzymamy:

```
3
5
4
```

Wiele argumentów funkcji

W Pythonie funkcje mogą przyjmować **wiele argumentów** w postaci rozpakowanej listy przy pomocy *****:

```
listaA = [3, 5, 7]
sumypar(*listaA)
(8, 12, 10)
```

Czasami liczba argumentów funkcji może nie być znana:

```
def iloczyn(*arg):
    ilocz = 1
    for x in arg:
        ilocz *= x
    return ilocz
```

```
iloczyn (2, 4, 5)          zwróci 40
iloczyn (2, 4, 5, 6)       zwróci 240
iloczyn (*range(6))        zwróci 120, czyli 5!
```

FUNKCJE REKURENCYJNE

Funkcja rekurencyjna to taka funkcja, które wywołuje samą siebie oraz posiada **warunek stopu**, który uniemożliwia takie wywołania w nieskończoność, np.:

```
def silnia(n):
    if n>1:   return n*silnia(n-1)
    else:     return 1
```

silnia(0) zwróci 1

silnia(5) zwróci 120

Większość funkcji iteracyjnych można zapisać w postaci funkcji rekurencyjnych,
lecz zwykle nie opłaca się to robić, gdyż każde wywołanie funkcji powoduje odkładanie
na stosie systemowym pewnej porcji danych (zmiennych), co sprawia,
iż takie funkcje działają wolniej niż w przypadku pętli iteracyjnej!

Ponadto niektóre funkcje rekurencyjne mogą wielokrotnie wykonywać te same obliczenia, np. w przypadku wyznaczania n-tego elementu ciągu Fibonacciego:

```
def fibonacci(n):
    if n<2:  return n
    else:  return fibonacci(n-1) + fibonacci(n-2)
```

Ilość wywołań funkcji rośnie w każdym kroku dwukrotnie, więc dla n=50 mamy ich 2^{49}.

Obliczenie takiej funkcji zajęłoby przeciętnemu komputerowi więc kilka miesięcy!

FUNKCJE ANONIMOWE

Jeśli funkcję zamierzamy wykorzystać tylko w jednym miejscu w programie, możemy wykorzystać wyrażenie **lambda argumenty : wyrażenie**
które jest odwzorowaniem przestrzeni danych w przestrzeń wyników, z które możemy od razu wykonać:

```
print (lambda x,y: x**y)(4, 0.5)                    # zwróci pierwiastek z 4, czyli 2
```

Wadą wyrażeń **lambda** jest niemożność wykorzystania w nich instrukcji nie będących wyrażeniami, np. print, if, for, while. Wtedy trzeba skorzystać z funkcji.

SŁOWNIKI

Słowniki – nazywane też tablicami asocjacyjnymi wiążą ze sobą dwie wartości: wartość klucza z wartością przechowywanej danej, podobnie jak w słowniku.

Słowniki od tablic, list, krotek i sekwencji różni to, iż są one indeksowane wartościami kluczy, a nie kolejnymi liczbami naturalnymi.

Słownik jest więc zbiorem par (klucz, wartość).

Klucze muszą być **unikalne**, gdyż każdemu kluczowi przypisana jest jakaś wartość.

Klucz musi być dowolnym obiektem niezmiennego typu

Słownik odwzorowuje więc obiekty dowolnego typu niezmiennego na inne obiekty dowolnego typu. Odwzorowanie to jest z punktu widzenia szybkości działania słownika jednostronne, aczkolwiek można szukać klucza odpowiadającego wartości.

telefonyalarmowe = {'pogotowie': 999, 'straz': 998, 'policja': 997}   # nawiasy {}

telefonyalarmowe ["pogotowie"]   zwróci 999

Słownik można rozszerzać o nowe pary elementów:

telefonyalarmowe ["pogotowie gazowe"] = 800 111 992

Możemy też modyfikować istniejące wartości:

telefonyalarmowe [„policja"] = 112

Słownik aktualnie utworzonych zmiennych zwraca funkcja **vars()**

python OPERACJE NA SŁOWNIKACH

Zawartość jednego słownika możemy przekopiować do innego słownika:

telefony = {'taxi': 9622, 'dom': 123456789}

telefondomowy = {'dom': 129876543}

telefonyalarmowe = {'pogotowie': 999, 'straz': 998, 'policja': 997}

mojetelefony = telefony.**copy**() # tworzy nowy słownik, do którego kopiuje pary

del telefony['taxi'] # usuwa parę wskazaną przez klucz

telefony.**clear**() # czyści cały słownik usuwając wszystkie pary

del telefony # usuwa cały słownik

telefony.**update**(telefondomowy) # aktualizuje wartości w słowniku wg innego słownika na podstawie kluczy

Otrzymamy: {'dom': 129876543, 'taxi': 9622}

telefony.**pop**('taxi') # ta funkcja łączy operację odczytu wartości z usunięciem pary o tym kluczu ze słownika po odczytaniu wartości

telefonyalarmowe.**popitem**() # usuwa ostatnią parę ze słownika po jej odczycie

Otrzymamy krotkę: ('policja', 997)

OPERACJE NA SŁOWNIKACH

telefonyalarmowe.**keys**() # zwraca listę kluczy: ['pogotowie', 'straz', 'policja']

telefonyalarmowe.**values**() # zwraca listę wartości: [999, 998, 997]

'taxi' **in** telefony # W celu sprawdzenia, czy określony klucz występuje w słowniku.

telefony.**has_key**('taxi') # Działa tak samo z wykorzystaniem funkcji has_key

997 **in** telefony.values() # W celu sprawdzenia, czy określona wartość występuje w słowniku.

Określenie ilości par w słowniku:

len(telefonyalarmowe)

Konwersja słownika na łańcuch (napis):

str(telefonyalarmowe)

Zwróci: "{'pogotowie': 999, 'straz': 998, 'policja': 997}"

ZBIORY

W Pytonie możemy tworzyć **zbiory zmienne** i **niezmienne:**

A = **set**([1,2,3,4,5]) # zbiór zmienny

B = **frozenset**([6,7,8,9]) # zbiór niezmienny

C = () # zbiór pusty

Do **zbiorów zmiennych** możemy dodawać nowe elementy lub je usuwać:

A.add(6) Otrzymamy: set([1,2,3,4,5,6])

A.discard(1) Otrzymamy: set([2,3,4,5,6])

Zbiory niezmienne mogą być kluczami w słownikach:

Dictionary={B:'szyfr'} Otrzymamy: {**frozenset**([6,7,8,9]):'szyfr'}

lub elementami innych zbiorów:

C.add(B) Otrzymamy: set([frozenset([6,7,8,9])])

Liczbę elementów zbioru zwraca funkcja **len**:

len(A) Otrzymamy: 5

len(C) Otrzymamy: 1

ZBIORY

Sprawdzenie, czy dany obiekt **jest elementem zbioru:**

2 in A Otrzymamy: true

lub nim nie jest:

7 not in B Otrzymamy: true

Sprawdzenie, czy dany zbiór jest **podzbiorem** innego:

C <= A

lub

C.**issubset**(A)

Sprawdzenie, czy dany zbiór jest **nadzbiorem** innego (zawiera inny zbiór):

C >= A

lub

C.**issuperset**(A)

Operacje mnogościowe na zbiorach:

D = A|B # łączenie zbiorów

D = A&B # przecięcie zbiorów (część wspólna zbiorów)

D = A-B # różnica zbiorów, typ zbioru wynikowego jest typem zbioru A

D = A^B # różnica symetryczna zbiorów (wzajemna)

REKORDY I KLASY

Rekordy w Pythonie mogą ulegać zmianom w trakcie działania programu, więc możemy je definiować jako puste i uzupełniać je w trakcie działania programu. Najpierw definiujemy **klasę** reprezentującą **rekord**, np.:

class Adres: pass

Potem tworzymy zmienną (obiekt) typu tej klasy:

a=Adres()

i dodajemy do niego kolejne pola:

a.miasto="Krakow"

a.kod="30-059"

a.ulica="al. Mickiewicza 30"

Odczytanie wszystkich pól rekordu, które posiadają wartości różne od domyślnych uzyskamy poprzez: a.**__dict__**

Otrzymamy: {'kod': '30-059', 'miasto': 'Krakow', 'ulica': 'al. Mickiewicza 30'}

Rekordy możemy tworzyć na podstawie innych rekordów:

```
class Wykladowca(Adres):
    imie = "Adrian"
    nazwisko = "Horzyk"
```

WYTWORNIKI LIST

Generatory list (list comprehensions) – tworzą bardziej skomplikowane listy:

- **Proste**: [wyrażenie **for** zmienna **in** sekwencja]
- **Proste warunkowe**: [wyrażenie **for** zmienna **in** sekwencja **if** warunek]
- **Rozszerzone**: [wyrażenie **for** zmienna1 **in** sekwencja1
  for zmienna2 **in** sekwencja2 ...]
  tworzona na podstawie więcej niż jednej listy
- **Rozszerzone z jednym warunkiem**:
  [wyrażenie **for** zmienna1 **in** sekwencja1
  for zmienna2 **in** sekwencja2 ...
  if warunek]
  pozwala na określenie warunku, który muszą spełniać dane listy wynikowej.
- **Rozszerzone z wieloma warunkami**:
  [wyrażenie
  for zmienna1 **in** sekwencja1 **if** warunek1
  for zmienna2 **in** sekwencja2 **if** warunek2
  ...]
  pozwala na określenie warunków, które muszą spełniać dane pobierane z poszczególnych list źródłowych, na podst. których tworzona jest lista wynikowa.

WYTWORNIKI LIST

Przykłady list generowanych:

```
listapoteg = [(x,x*x) for x in range(1,10)]

listakodowASCII = [(x, ord(x)) for x in "ABCDEFGH"]

listapodzielnychprzez3 = [x for x in range(1,100) if not (x%3)]

listawsporzednych = [(x,y) for x in range(1,10,2) for y in range(5,1,-1)]

listaparmniejszywiekszy = [(x,y) for x in range(1,10) for y in range (10,1,-1) if x<y]

lista = [(x,y) for x in range(1,20) if x%2
        for y in range (20,1,-1) if not (y%3)]
```

PRZETWARZANIE SEKWENCJI

Sekwencje danych można przetwarzać z wykorzystaniem specjalnych funkcji:

- **apply** – służy do rozpakowania sekwencji danych (podobnie jak gwiazdka)
- **map** – wywołuje określoną funkcję dla każdego elementu sekwencji z osobna i zwraca listę rezultatów o liczebności równiej ilości parametrów w liście
- **zip** – służy do konsolidacji danych, czyli łączenia kilku list w jedną, przy czym wartość wynikowa (krotka) zależy od wartości elementów list źródłowych
- **filter** – wykonuje filtrację danych sekwencji danych, dla który funkcja zwraca true
- **reduce** – wykonuje operację agregującą na danych sekwencji jako parametrach funkcji, zwracając pojedynczą wartość

srednia=lambda x,y: (x+y)/2   zwróci: 3 dla wywołania srednia(2,4)

xy = (2,4)   # definiujemy krotkę (sekwencję)

apply(srednia, xy)   zwróci również: 3

map(lambda x: x*x, range(1,10))   zwróci listę: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

map(srednia, range(5), range(6,11))   zwróci listę: [3, 4, 5, 6, 7]

zip(range(1,5), range(4,0,-1))   zwróci listę krotek: [(1,4), (2,3), (3,2), (4,1)]

filter(lambda x: x.lower() in "aeiouy", "lubie przychodzic na wyklady")   zwróci: 'uieyoiayay'

reduce(lambda x,y: x*y, range(1,5))   zwróci 24   # 24 = (1 * 2 * 3 * 4)

reduce(lambda x,y: x+y, map(lambda x: x*x, range(1,20)))   # oblicza sumę kwadratów

PLIKI

Możliwość zapisania danych na stałym nośniku dają nam **pliki**:

plik1 = **open**("plik.txt", „rb") # Otwiera plik do odczytu w trybie binarnym

plik1 = **open**("plik.txt", "wb") # Otwiera plik do zapisu w trybie binarnym

plik1 = **open**("plik.txt", „ab") # Otwiera plik do dodawania na końcu nowych elementów w trybie binarnym „b"

plik1 = **open**("plik.txt", „r+b") # Otwiera plik do modyfikacji w trybie binarnym

Obiekty plikowe posiadają trzy podstawowe atrybuty:

plik1.**name** - zawiera nazwę pliku

plik1.**mode** - określa tryb otwarcia pliku: w – do zapisu, r – do odczytu

plik1.**closed** - określa, czy plik jest zamknięty

Obiekty plikowa mają wbudowane funkcje:

plik1.**write**("Zapisz tą linię do pliku.\n")

Chcąc zmusić system do opróżnienia bufora i zapisania wszystkich danych do pliku:

plik1.**flush**()

Na końcu pracy z plikiem należy go zamknąć:

plik1.**close**()

OPERACJE NA PLIKACH

plik1.**read()** - odczytuje linię z pliku

plik1.**read(n)** - przeczytanie fragmentu pliku o podanej długości n

plik1.**write(**"Linia do zapisu"**)** - zapisuje linię w pliku

plik1.**readlines()** - wczytuje z pliku sekwencję linii dodając na końcu każdej znak '\n'

plik1.**writelines ()** - zapisuje do pliku sekwencję linii nie dodając separatorów

plik1.**tell()** - podaje aktualną pozycję w pliku

plik1.**seek(n)** - ustawia pozycję odczytu w pliku na podaną wartość

plik1.**seek(k,1)** - przesunięcie względne w pliku o k pozycji, k może być ujemne

plik1.**truncate(n)** - obcina plik od podanej pozycji

Szyfrowanie danych z wykorzystaniem szyfru Cezara:

```
nazwapliku = raw_input("Podaj nazwę pliku do zaszyfrowania: ")
przesuniecie = input("Podaj przesunięcie (o ile znaków): ”)
file1 = open(nazwapliku,"r+b")                # otwieramy plik do odczytu i zapisu (modyfikacji)
zawartosc = file1.read()                      # wczytujemy do zmiennej zawartosc tekst źródłowy
zawartosczaszyfrowana=""
for c in zawartosc:                           # dla każdego znaku w zawartosc
ascii=ord(c)                                  # wyliczamy kod ASCII
if a > 32:                                    # znaków białych i sterujących nie szyfrujemy
    znak=chr((ascii+ przesuniecie) % 256)     # inne przesuwamy
    zawartosczaszyfrowana +=znak              # dodajemy do zawartosczaszyfrowana
file1.seek(0)                                 # przesuwamy aktualną pozycję w pliku na jego początku
file1.write(sz)                               # zapisujmy zaszyfrowaną zawartość do pliku
file1.close()                                 # zamykamy plik
```

python OPRACJE NA KATALOGACH

getcwd() # podaje nazwę aktualnego katalogu na dysku

chdir('nazwakatalogu'**)** # zmienia nazwę aktualnego katalogu na dysku

listdir('nazwakatalogu'**)** # podaje zawartość aktualnego katalogu na dysku

mkdir('nazwakatalogu'**)** # tworzy katalog o podanej nazwie w aktualnym katalogu

path.exists('sciezkaobiekt'**)** # sprawdza, czy dany obiekt dyskowy istnieje

rename('nazwastara', 'nazwanowa'**)** # zmienia nazwę pliku lub katalogu

path.getsize() # zwraca długość pliku w bajtach

Przykład: NWD i NWW

```
# Wyliczanie NWD i NWW metodą Euklidesa
# 1. wprowadzanie liczb
print "Podaj dwie liczby naturalne:”
a = input("Pierwsza liczba:")
b = input("Druga liczba:")
# 2. ustalenie, która spośród nich jest mniejsza
if a > b:
        w = a; m = b
else:
        w = b; m = a
# 3. pętla główna
r = w % m
while r:
        w = m; m = r; r = w % m
# 4. wyświetlanie i formatowanie rezultatów
print "NWD liczb %i i %i wynosi %i, a ich NWW wynosi %i" % (a, b, m, a*b/m)
```

Przykład: Liczby pierwsze

```
# Wyszukiwanie liczb pierwszych sitem Eratostenesa
koniec = input("Podaj zakres poszukiwania liczb pierwszych:")
pierwsze = range(koniec+1)
n = 1
while n < koniec:
	n += 1
	if pierwsze[n]==0:		# nie bierzemy pod uwagę liczby nie pierwsze
		continue
	m = n * 2
while m <= koniec:
	pierwsze[m]=0		# zerem oznaczamy liczby, które nie są pierwsze
	m += n
print "Znaleziono następujące liczby pierwsze:"
for n in pierwsze[2:]:
	if n: print n
```

LITERATURA

Literatura w języku polskim:

- David M. Beazley: Programowanie: Python. Read Me 2002, ISBN 83-7243-218-X.
- Chris Fehily: Po prostu Python. Helion 2002, ISBN 83-7197-684-4.
- Mark Lutz: Python. Leksykon kieszonkowy. Helion 2001, ISBN 83-7197-467-1.
- Marian Mysior (tłum.): Ćwiczenia z... Język Python. Mikom 2003, ISBN 83-7279-316-6.
- Wydania specjalne czasopism: Software 2.0 Extra! 9: Poznaj moc Pythona!
- M. Lutz, Python. Wprowadzenie, 3. wydanie, Helion, 2009.
- Python. Od podstaw, Helion 2006.
- G. Wilson, Przetwarzanie danych dla programistów, Helion, 2006.
- J. E. F. Friedl, Wyrażenia regularne, Helion, 2001.

Literatura w języku angielskim:

- Michael Dawson: Python Programming for the Absolute Beginner. Premier Press 2003.
- P. Barry, D. Griffiths, Head First Programming, O'Reilly Media, 2009.
- Mark Lutz: Programming Python, 3nd Edition. O'Reilly 2006.
- Alex Martelli: Python in a Nutshell. O'Reilly 2003.
- David Ascher, Mark Lutz: Learning Python, 2nd Edition. O'Reilly 2003.
- M. Lutz, Learning Python: Powerful Object-Oriented Programming, 4th Edition, O'Reilly Media, 2009.

LITERATURA

Strony internetowe w języku polskim i angielskim:

- Polskie tłumaczenie dokumentacji: http://www.python.org.pl/
- Przykłady kodów źródłowych: http://python.kofeina.net/
- Kurs Pythona: http://www.mychkm.webpark.pl/python/
- Kursy Pythona: https://pl.python.org/kursy,jezyka.html
- Kurs Pythona: http://openbookproject.net/thinkcs/python/english3e/index.html
- Kurs edukacyjny Pythona: http://python.edu.pl/

Literatura angielskojęzyczna dostępna w sieci:

- A. Downey, Think Python. How to Think Like a Computer Scientist, Green Tea Press, 2008: http://www.thinkpython.com
- Dive Into Python: http://diveintopython.org/
- Dive Into Python 3: http://diveintopython3.org/
- Learning with Python 2nd Edition: http://openbookproject.net/thinkCSpy/
- A Byte of Python v1.92 for Python 3.0: http://www.swaroopch.com/notes/Python
- Advanced Scientific Programming in Python: http://portal.g-node.org/python-summerschool-2009/

Oficjalne strony Pythona:

- http://python.org/
- http://docs.python.org/
- http://www.python.org.pl/

Ściągnięcie kompilatora Pythona:

http://www.python.org/download/

Spider: https://github.com/spyder-ide

PyCharm: https://www.jetbrains.com/pycharm/