Python dla początkujących

Python to popularny (gdyż prosty, o dużych możliwościach, a zarazem darmowy) język skryptowy, opracowany we wczesnych latach 90. przez Guido van Rossuma. Obecna wersja to 2.6, ściągnij z python.org i zainstaluj.

W Pythonie można pracować w trybie interaktywnym lub skryptowym. Ten pierwszy jest bardzo poręczny do nauki języka – wpisujemy polecenie i od razu widzimy efekt. Uruchom edytor (a właściwie małe zintegrowane środowisko języka) IDLE i wpisz: 3 + 2 (klepnij Enter). Działa?

Ten sam efekt można otrzymać pisząc: print 3+2. A także print(3+2). Jednak w trybie interaktywnym na ekranie jest wyświetlana wartość ostatniego wyrażenia i słowo print nie jest koniecznie.

Arytmetyka, typy danych

Jednym z pierwszych, najprostszych zastosowań dowolnego języka programowania jest użycie go jako kalkulatora. Python nadaje się do tego celu znakomicie.

Wypróbuj operatory: +, -, *, /, także z nawiasami (okrągłymi), dowolnie zagnieżdżonymi. Wszystko jasne? 2 ** 3 – daje 8, więc zgadujesz, że chodzi o potęgowanie. No to teraz wpisz: 11 ** 123. Duża liczba! W C, C++, Pascalu czy Javie nie poszłoby Ci tak łatwo z tym obliczeniem (w bibliotece standardowej Javy jest wprawdzie klasa BigInteger, ale daleko jej do wygody Pythona). Oczywiście operacji na ogromnych liczbach nie wykonuje się często, ale są sytuacje, gdy się one przydają (np. implementacja algorytmów kryptograficznych).

A teraz sprawdź 5 / 3. Wyszło jak np. w C, prawda? Żeby otrzymać wynik „poprawny”, napisz 5.0 / 3. Albo 5./3, albo 5/3.0 etc. Zgadujesz na czym polega różnica, ale przekonaj się naocznie: wpisz type(5). A teraz type(5.0). Wszystko jasne?

Nie tylko stałe / literały mają typy, ale także zmienne **w danej chwili** przechowują wartości określonego typu. Napisz:

```
x = 5
print type(x)   # albo samo type(x)
```

(po # jest komentarz, do końca wiersza)

x jest zmienną, która w danej chwili przechowuje wartość 5, i jest oczywiście ona typu *int*. Możesz przy okazji sprawdzić np. ile wynosi 2*x.

A teraz wpiszmy:

```
x = 3.1
```

i znów sprawdźmy typ. Zmienił się. Czyli zmienna x jest tylko niejako referencją, etykietą do miejsca w pamięci, które raz może przechowywać liczbę całkowitą, a raz zmiennoprzecinkową, a może jakiś string, albo obiekt dowolnego typu itd.
Wypróbuj zatem:

```
x = "Abc"
x = 3 + 4j  // liczba zespolona!  Typ "complex"
```

```
x = [4, -1.5, "e"]  // lista
x = 5 ** 43
```

I w każdym z tych przypadków sprawdź typ x. Co było w ostatnim przypadku? Powinien być typ "long" (jedyne typy całkowite w Pythonie to int i long). Wyświetl wartość x. Dostrzegasz literkę L?

Wypróbuj jeszcze operator % (modulo oczywiście) i dzielenie przez zero (jaki efekt? Wyjątek!). Liczby zespolone: wykonaj działania, np. odejmowanie, mnożenie, dzielenie. Te same symbole, tylko argumenty typu zespolonego. Podstaw pod np. a liczbę 2 + 4j, wypisz a* (3-2.5j).

Python (ściągając z języka C) ma też wbudowaną obsługę systemu ósemkowego i 16-kowego. Czyli 071 to będzie 1*1 + 7*8 = 57, natomiast 0x3A1 to będzie 1*1 + 10*16 + 3*256 = 929. Wypróbuj. Sprawdź też błędne literały, np. 085.

Jak działa instrukcja podstawienia (przypisania), to już wiesz. Można też podstawiać hurtowo (a = b = c = 3), a co ciekawsze można działać na tzw. krotkach (więcej o nich później), dzięki czemu możliwe są zapisy typu:

```
x, y = -5, "Ala ma kota"
```

(sprawdź, co jest pod x i y),
a także

```
x, y = y, x
```

co nieraz przecież bardzo wygodne.
Oczywiście krotki mogą być „dłuższe”, np.:

```
p1, p2, p3, p4, p5 = 2, 3, 5, 7, 11
```

Znając C, odgadniesz jak działają operatory +=, *=, /= etc. (ale wypróbuj).

Z ciekawostek: w trybie interaktywnym znak _ (podkreślenie) przywołuje wartość ostatniego obliczonego wyrażenia.

Napisy (stringi)

Napisy są typu *str*. Można je podawać w cudzysłowach podwójnych lub pojedyńczych, stąd zapisy:

```
powitanie = "dzień dobry"
powitanie = 'dzień dobry'
```

są równoważne.
Ale:

```
powitanie = 'dzień dobry"
```

jest błędem (trzeba być konsekwentnym).
Można również używać napisów wielolinijkowych:

```
s = """Ten napis
dłuuugi jest
..."""
```

Jak zapisać w stringu: Wielokrotnie oglądałem film "Sami swoi". ?
Są przynajmniej dwie możliwości:

s = 'Wielokrotnie oglądałem film "Sami swoi".'
s = "Wielokrotnie oglądałem film \"Sami swoi\"."
Wybór jest kwestią gustu.
Możesz w napisach używać znaków sterujących \n i \t. Jak wprowadzić backslash też powinno być oczywiste: użyj \\.

Teraz sprawdź jak wyglądają: s[0], s[2], s[100], len(s), s[-1]. Widzisz, co oznacza to ostatnie? To może jeszcze s[-2].
A także: s[1:4], s[0:4], s[:4], s[7:], s[::2], s[::-1].

Możliwe są konwersje liczby na string i w drugą stronę. Wypróbuj:
s = "125"
print 2*int(s)

Albo:
n = 3
s = "Ala ma " + str(n) + " koty"
print s

W drugim przypadku dodatkowo widzieliśmy konkatenację stringów (operator +). Działa również +=. Bardziej zaskakującym operatorem dla stringów jest *:
print "*" * 70
"ab " * 8

W Pythonie nie ma osobnego typu dla pojedyńczych znaków, tj.
print type(s[0])
oraz
s1 = "a"
print type(s1)
wypiszą "str", ale dla stringów 1-znakowych działa funkcja chr(). Przykład: chr(65) – zwróci string "A". Jak zgadujesz, 65 to kod tej litery w ASCII. Nie musisz mi wierzyć – możesz sprawdzić w taki sposób:
ord("a")
Oczywiście ord("ab") rzucałoby wyjątek (TypeError).
(Przy okazji: w Pythonie 3.x napisy są już w Unicodzie, ale ponieważ standardowe znaki ASCII są w nim „zanurzone”, wynik funkcji ord("a") i tam byłby taki sam.)

Powiedzmy, że s == "kosa". Chcemy drugi znak zmienić na 'a'. s[1] = 'a' ? Niestety, nie zadziała. Stringi w Pythonie są *niemutowalne* (niezmienne). Trzeba napisać:
s = s[0] + 'a' + s[2:]
Stringi są obiektami. Składnia wywołania metody obiektu przypomina np. C++ (z kropką). Oto przykłady metod, które możemy wywołać dla napisu s:
s.capitalize() – zwraca napis ze zmienioną pierwszą literą na wielką
s.isdigit() – sprawdza, czy wszystkie znaki są cyframi
s.islower() – sprawdza, czy wszystkie litery są małe
s.center(długość) – centruje napis w polu o podanej długości (uzupełniając spacjami)
s.rjust(długość) – wyrównuje do prawej w polu o podanej długości (uzupełniając spacjami)
s.count(s1) – zlicza wystąpienia podciągu s1 w s
s.lstrip() – zwraca napis z usuniętymi wiodącymi białymi znakami

Podkreślmy: funkcje te nie zmieniają s! Np. s.center(...) zwraca nowy napis. Przypominamy też wielce użyteczną funkcję len(s) (wykorzystywana w Pythonie do rozmaitych kolekcji, nie tylko stringów).

Wprowadzanie danych

```
imie = raw_input("Podaj swoje imię ")
print "Witaj ", imie, ", widzę, że nieźle Ci idzie!"
```

Funkcja raw_input zwraca string. Ale istnieje również funkcja input, zwracająca liczbę (*int*, *long*, *float* lub *complex*) lub wartość logiczną (True albo False, typ *bool*). Można też wprowadzić naraz krotkę:

```
a, b = input("podaj 2 liczby (oddzielone przecinkiem) ")
print a, b
```

Zauważ: przecinek rozdziela argumenty print-a pojedyńczą spacją. Taki kod:

```
print a
print b
```

wstawiłby znak nowego wiersza zarówno po a, jak i po b.

Instrukcja warunkowa...

...nazywa się w Pythonie (zgadłeś!) "if", a składnia jest taka:

```
if warunek:
   # zrób coś
elif warunek:
  # tu też zrób coś
else:
  # ewent. zrób coś w takim przypadku
```

Oczywiście bloku else ani też elif być nie musi. "elif" to skrót od "else if" i znaczy dokładnie to samo. W warunkach można używać operatorów koniunkcji (and), alternatywy (or) czy negacji (not).
Przykład:

```
if x > 3 and y == -1 and 2 < z < 4:
  print …
  …
else:
  …
```

Jak widzisz, są podobieństwa, ale i różnice w stosunku do języka C. Np. użyteczne nierówności obustronne.

BARDZO WAŻNA UWAGA. W Pythonie bloków nie oznaczamy klamerkami {} ani też np. słowami begin...end, a jedynie WCIĘCIAMI z lewej strony. Wolno używać do tego celu spacji i

tabulatorów, ale odradza się ich mieszanie. Wszystkie linie w danym bloku muszą mieć takie samo wcięcie.

Zadania:

1. Poproś użytkownika o podanie dwóch napisów, przy czym drugi z nich musi być 1-literowy (jeśli nie jest, wypisz komunikat o błędzie). Następnie napisz, ile razy drugi napis mieści się w pierwszym. Przykład: ananas, a → 'a' mieści się 3 razy w 'ananas'
2. Poproś użytkownika o podanie napisu s o długości przynajmniej 20, a nie więcej niż 30 (w razie potrzeby wypisz komunikat o błędzie), a następnie utwórz string będący 10-krotnym powieleniem litery środkowej s (czyli np. przy długości 23 będzie nią s[11]).

Listy

Listy w Pythonie to w zasadzie tablice dynamiczne (tj. takie, o których rozmiar nie musisz się kłopotać, bo w razie potrzeby „same" się powiększą). Co ciekawe, elementy listy nie muszą być tego samego typu. Przykład:

```
varia = ["abc", 31, -3.14, 2+4j, [2, 1]]
print varia[2]   # -3.41
print varia[-1]   # [2, 1]
print type(varia)  # 'list'
```

Jak widać z przykładu, elementem listy może być nawet inna lista. De facto, jest to standardowy sposób otrzymywania tablic 2- i więcej wymiarowych. Zgadnij, co wyświetli print varia[-1][0].

```
lista1 = []  # ta lista jest pusta
len(lista1)  # wyświetli 0
varia[1:3]  # wyświetli [31, -3.14]
varia[1::2]  # co drugi element poczynając od varia[1]
```

Listy można powielać, np. varia *= 2, można doklejać inną listę na końcu:

```
lista1 += lista2  # zakładając, że lista2 istnieje i jest typu "list"
```

albo też tak:

```
lista1.expand(lista2)
```

Można wyrzucać element(y) z listy:

```
del lista1[1:3]  # wylecą elementy na pozycjach 1 i 2
del lista[-2]  # wyleci przedostatni
```

Uwaga: usuwanie elementu, który nie jest blisko końca listy, jest powolne!

Typowa operacja to doklejenie elementu na końcu:

```
x = 2
lista1.append(x)
```

albo:

```
lista1 += [x]
```

Listy można porównywać przy użyciu ==, !=, a nawet >, >=, <, <= (sprawdź działanie!).

Odpowiedz też, czym się różni metoda append od extend. Czy argumentem append może być lista?

Można szybko sprawdzić, czy dany element należy do listy:
31 in varia
Otrzymamy odpowiedź True, bo istotnie, element 31 należy do listy varia.
Można też pisać w stylu: 5 not in varia.

Do list jeszcze wrócimy, ale teraz...

Pętla for

Trudno wyobrazić sobie programowanie bez pętli. W Pythonie zdecydowanie najczęściej używaną jest pętla for:

```
for i in range(10):
  print i,
```

wypisze liczby od 0 do 9 oddzielone spacjami. Co znaczy range? Generuje ono zakres, tj. listę złożoną z liczb naturalnych tworzących szereg arytmetyczny. Są 3 warianty range:

```
range(n)   # n > 0, lista [0, 1, 2, ..., n-1]
range(m, n)  # m < n, lista [m, m+1, …, n-1]; jeśli m >= n, to lista pusta
range(m, n, step)  # jak wyżej, ale co step wartości
```

Np.:

```
range(0, 12, 2)  # [0, 2, 4, 6, 8, 10]
range(1, 7)  # [1, 2, 3, 4, 5, 6]
```

Powyższy opis nie jest kompletny, można i tak:

```
range(5, 1, -1)  # [5, 4, 3, 2]
range(8, -1, -2)  # [8, 6, 4, 2, 0]
```

A teraz iterowanie po liście:

```
for i in varia:
  print i
```

A może z pominięciem pierwszego elementu? Proszę:

```
for i in varia[1:]:
  print i
```

Można iterować po stringu (który też jest „sekwencją”):

```
s = "Witaj!"
for i in s:
  print i,
```

Z pętli można szybciej wyjść poprzez break (znów „inspiracja” z języka C):

```
for i in range(20):
```

```
  print i,
  if i % 5 == 4:
    break
```

Odpowiedz, ile liczb pojawi się na ekranie.

Zadania:

3. Wygeneruj na ekranie tabliczkę mnożenia 10 x 10 (nie musi wyglądać ładnie, ale niech w każdym wierszu będzie dokładnie 10 liczb). Wskazówka: pętle możesz zagnieżdżać.
4. Znajdź i wypisz na ekranie wszystkie liczby pierwsze z przedziału [20, 100].
5. Policz 15!, wypisz na ekranie tylko 3 pierwsze cyfry tej liczby.
6. Wczytaj liczbę całkowitą od użytkownika i utwórz z niej string wstawiający kropki co 3 cyfry od końca (zastosowanie: finansowe, np. "1.600.000" wygląda czytelniej niż 1600000).

Pętla while

To chyba dobra wiadomość: pętla while jest drugą i ostatnią pętlą w Pythonie. Warunek sprawdzany jest na początku:

```
x = -1
while x < 3:
  print x,
  x += 1
print "koniec"
```

Jak zatem „zasymulować" pętlę do...while, znaną z C (czy jej odpowiednik repeat...until z Pascala)? Aby warunek sprawdzany był na końcu?

Standardowy chwyt wygląda następująco:

```
while True:
  ...
  if warunek_wyjścia:
     break
# ciąg dalszy programu
```

Przykład: chcemy zsumować odwrotności kolejnych liczb naturalnych, aż suma przekroczy 5. Kod:

```
j = 0
suma = 0.0
while True:
  j += 1
  suma += 1/float(j)   # można i tak zamiast np. 1./j
  if suma > 5:
    break
print "zsumowano", j, "składników, suma wynosi", suma
// Wyszło niezbyt po polsku: "83 składników", ale nie będziemy się teraz tym martwić
```

Skoro już używamy while True, to omówmy...

Typ logiczny (bool)

Typ ten ma dwie wartości: True i False (zwróć uwagę na wielkość liter). Warto jednak wiedzieć, że jako False ewaluowane są następujące wyrażenia:
None
0
"" # czy inaczej zapisane łańcuchy puste
()
[]
{}

To nic, że nie ze wszystkimi z w/w się już spotkałeś. Jeszcze wiele przed nami.

Do pewnego stopnia przypomina to konwersje w C/C++. Sprawdź:
bool(3), bool(0), bool(0.0001), bool(0.0), bool(-2), bool(3 > 4).

Co otrzymasz na ekranie? Najpierw pomyśl, potem sprawdź:

```
li = [2,3,1]
while li:
  print li[-1],
  del li[-1]
```

Można się domyślić, że jeśli np. napis pusty ewaluowany jest jako False, to dowolny napis niepusty będzie True. Sprawdź:
bool("Python jest fajny")

A nawet takie coś jest możliwe (choć nie polecam); pomyśl najpierw, co wyjdzie:
print 10 + True + False + True + (2==2)

Oczywiście wyrażeń logicznych używa się nie tylko przy while, ale też przy if i elseif.

Teraz taki przykład (inspiracja z książki „Beginning Python" M. L. Hetlanda). Najpierw pomyśl, czy **dokładnie** rozumiesz, co się dzieje, potem uruchom.

```
name = raw_input("Your name, sir/madam? ")
while not name:
  print "You must give some name"
  name = raw_input("Your name, sir/madam? ")
if name.title() != "Bill Gates":
  print "Welcome!"
else:
  print "Access denied"
```

Krotki (typ tuple)

Krotka przypomina listę, tyle że jest niemutowalna (niezmienialna). Składnia:

```
kolor = (128, 0, 255)   # nawiasy okrągłe
```

Próba zmiany jakiejś składowej nie powiedzie się:

```
kolor[1] = 20
```

Jeżeli więc chcesz zmienić krotkę, to powołaj do życia nowy obiekt (o tej samej nazwie):

```
kolor = (128, 20, 255)
```

(Przy okazji: stary obiekt kolor zostanie wkrótce „w tle” (i bez Twojej wiedzy) usunięty przez odśmiecacz pamięci (*garbage collector*).)

Po krotce można iterować:

```
for skladowa in kolor:
  print skladowa,
```

Można ją „rozpakowywać”:

```
r, g, b = kolor   # podstawienie hurtowe
```

Tudzież robić parę innych rzeczy, ale nie da się ukryć, że krotka przypomina listę, ale jest jednak od niej mniej elastyczna. Po co więc w ogóle zadawać się z krotkami?

Powodów jest kilka. Najważniejszy chyba taki, że tylko typy niemutowalne (a więc m.in. krotka – ale już nie lista) mogą być kluczami w słownikach i zbiorach (o tych strukturach danych niebawem). Ponadto krotka może być (nieco) szybsza, a także zabezpieczać programistę przez pewnymi błędami.

Krotka może mieć tylko jedną składową; zapis wygląda nieco zaskakująco:

```
t = (12,)
```

Są funkcje do konwersji listy na krotkę i vice versa: ale uwaga, nie zmieniają one obiektu na którym działają, lecz zwracają nowy:

```
li = [2, 7, -3.5]
tu = tuple(li)
type(li)
type(tu)
```

Wypróbuj analogicznie funkcję list().

Przy okazji: listę możemy posortować (zmienia się bieżący obiekt), a krotki nie:

```
li.sort()   # wyświetl teraz li
tu.sort()  # AttributeError: 'tuple' object has no attribute 'sort'
```

Funkcje max, min i sum natomiast działają na krotkach i listach:

```
print max(tu)
print max(li[:-1])
print sum((4, 1, 12))  # podwójne nawiasy: wewnętrzne wskazują, że argumentem jest krotka
```

Słowniki

To jedna z fajniejszych rzeczy w Pythonie: bardzo wygodna w obsłudze tablica asocjacyjna (inne funkcjonujące nazwy: słownik, hash, mapa...). W odróżnieniu od zwykłych tablic (czyli np. znanej nam już listy w Pythonie), do elementów tablicy asocjacyjnej odwołujemy się przez klucz, który może być dość dowolnego typu. W zwykłej tablicy elementy były indeksowane kolejnymi intami – tu jest wolna amerykanka. Klasyczne zastosowanie to książka telefoniczna:

```
ksiazka_tel = {}   # te klamerki oznaczają słownik, typ dict
ksiazka_tel["Hamlet"] = 619433156
ksiazka_tel["Ofelia"] = 539991474
print ksiazka_tel["Ofelia"]
print ksiazka_tel["Horacy"]  # klucz nieznany!
ksiazka_tel["Borewicz"] = 997

for (k, v) in ksiazka_tel.items():    #  k-key, v-value (oczywiście można nazwać inaczej)
  print k, "ma telefon o numerze", v
```

Ale najprościej wyświetlić cały słownik po prostu tak:
```
ksiazka_tel  # albo print ksiazka_tel, jeśli nie w trybie interaktywnym
```

Borewicz dostał służbową komórkę? No to zmieniamy:
```
ksiazka_tel["Borewicz"] = 617555321
print ksiazka_tel["borewicz"]  # ops, błąd!  Małe i duże litery rozróżnialne
```

Sprawdźmy, czy w międzyczasie Horacy nie dorobił się telefonu:
```
  if "Horacy" in ksiazka_tel:
    print "Już jest!  Jego numer to", ksiazka_tel["Horacy"]
```

Tu nam się nie udało, ale zmieńmy Horacego np. na Hamleta.

Czy w obrębie jednego słownika mogą być klucze (a także skojarzone wartości) różnych typów? Pachnie szaleństwem, ale tak!
Wypróbuj przykłady, żebyś dobrze zapamiętał, czego nie powinieneś robić: ☺
```
ksiazka_tel[(255,0,0)] = "czerwony"
ksiazka_tel["W tym szaleństwie jest metoda"] = True
ksiazka_tel[608] = [60801, 60804, 60802, 60899]
```

Czasem od razu chcemy słownik czymś zainicjować. Najprościej tak:
```
kolory = { "czerwony": (255,0,0), "niebieski": (0,0,255), "zielony": (0,255,0), "biały": (255,255,255) }
```

Powiedzmy, że mamy także słownik
```
kolory2 = {"czerwony": (200, 0,0), "czarny": (0,0,0)}
```
Chcemy zaktualizować stary słownik; robi się to tak:
```
kolory.update(kolory2)
```

Sprawdź, że nic ze słownika *kolory* nie wypadło, natomiast zmieniła się (została nadpisana) wartość skojarzona z kluczem „czerwony”, a także doszedł klucz „czarny”. Oczywiście *kolory2* się nie zmienił.

Oto przykłady dalszych metod, które możemy wywołać dla słownika *d*:
d.keys() – zwraca listę kluczy
d.values() – zwraca listę kluczy
d.clear() – czyści cały słownik (ale zmiennej d nie usuwa)
d.copy() – zwraca kopię *d* (d_kopia = d.copy() tworzy „niezależną" kopię, d_kopia = d byłoby tylko skopiowaniem referencji i każda zmiana *d* pociągałaby zmianę *d_kopia*)

Usunięcie klucza "abc":
```
del d["abc"]  # KeyError, jeśli próbujemy usunąć coś, czego w słowniku nie ma
```

Co jeszcze warto wiedzieć? Wartościami (ale nie kluczami) słownika mogą być nawet inne słowniki. Utwórzmy więc coś, co można by szumnie nazwać małą obiektową bazą danych:

```
znajomi = { "Wojtek": { "tel": 487133, "miasto": "Zgierz", "ulica": "Zachlapana 14" }, \
  "Ola": { "tel": 604111265, "miasto": "Opole", "ulica": "Krótka 1" }, \
  "Sebastian": { "tel": 608726122, "miasto": "Tarnowskie Góry", "ulica": "Knutha 34" } }
print znajomi["Ola"]["miasto"]
```

Zbiory

Zbiór (ang. *set*) przypomina słownik, tyle że do słowników wrzuca się pary (klucz-wartość), a do zbiorów pojedyńcze elementy. Zbiór jest kolekcją, w której elementy nie mogą się powtarzać, a ich kolejność nie jest ustalona. Innymi słowy, zbiory [1,3,2], [1,2,3], [3,2,1,2,3] są wszystkie identyczne.

```
zb = set()
zb.add(3); zb.add(1); zb.add(3)  # można zapodać wiele instrukcji w linii - ze średnikami
zb.add(-2)  # ale zaleca się pisać każdą instrukcję w osobnej linii
for i in zb:
  print i,

print zb[0]  # błąd!  TypeError: 'set' object does not support indexing
```

Zbiór można zainicjować listą:
```
zb = set([4, 1, 2, 10.4])
```

Dla zbiorów działają relacje: ==, !=, a także >=, <=, którymi sprawdzamy, czy jeden zbiór jest podzbiorem (nadzbiorem) drugiego, np.:
```
A = set([3, 1, 8])
B = set([1,8])
B <= A   # zwraca True
```

Mamy też równoważne metody *issubset*, *issuperset* (wypróbuj).

Oprócz dodawania elementów, można elementy usuwać (metoda *discard* lub *remove*; różnią się tylko przy próbie usunięcia elementu, którego nie ma w zbiorze – sprawdź jak!), działa też *len*. Przynależność do zbioru sprawdzamy za pomocą *in*, tj.:

```
3 in A  # zwróci True
8 not in B # zwróci False
```

Dostępne są operacje mnogościowe na zbiorach: suma (metoda *union* albo operator |, działają tak samo), przekrój (część wspólna) (metoda *intersection* albo operator &), różnica (metoda *difference* albo operator -) i różnica symetryczna (metoda *symmetric_difference* albo operator ^).
Przykłady:

```
C = (A - B).union([-1,2,4])
D = set([7,8]) & B
E = A.symmetric_difference(set([1,2,3,5]))
```

Istnieje też typ *frozenset*, tj. zbiór niemutowalny.

```
X = frozenset([5, 1, 2])
X.add(3)  # AttributeError: 'frozenset' object has no attribute 'add'
```

Obiekty typu *frozenset*, w przeciwieństwie do *set*, mogą być kluczami w słownikach oraz elementami innych zbiorów. Można też „zamrażać" i „rozmrażać", np. frozD = frozenset(D).

Moduły

Moduły to, z grubsza mówiąc, biblioteki funkcji, których można używać w swoich programach. Co ciekawe, postać pliku tworzącego moduł w zasadzie nie różni się od „zwykłego" pliku zawierającego skrypt. Często korzystamy z modułów biblioteki standardowej, a robi się to w taki sposób:

```
import math
print math.sqrt(7)
```

```
import string
spolgloski = set(string.ascii_lowercase) - set("aeiouy")
```

```
from random import *
print randint(1,3)
```

Ostatni przykład mógł wyglądać inaczej:

```
import random
print random.randint(1,3)
```

Można też importować funkcje indywidualnie:

```
from random import randint, sample
sample([1,2,4,8,10], 2)  # działa
choice([1,3,4])  # nie działa!
random.choice([1,3,4])  # też nie działa!
```

```
import random
random.choice([1,3,4])  # teraz działa!
```

Żeby podejrzeć listę dostępnych funkcji w danym, zaimportowanym już, module, użyjmy funkcji dir, np. dir(random). Nie zważamy na razie na funkcje postaci __cośtam__ (mają one

specjalne znaczenie). Jeśli interesuje nas konkretna funkcja, to krótką ściągę otrzymamy przy pomocy __doc__, np.: random.choice.__doc__ Bardzo użyteczne, korzystaj!

Do tworzenia własnych modułów dojdziemy niedługo... gdy będziemy umieli tworzyć funkcje.

Zadania:

7. Wczytaj dwie nazwy miesięcy (nie sprawdzamy poprawności pisowni) i napisz, ile miesięcy mija od pierwszego z miesięcy do drugiego. Jeśli drugi jest wcześniejszy, to zakładamy, że jest on w następnym roku. Przykład: marzec, maj, na wyjściu: 2; maj, styczeń, na wyjściu: 8; luty, luty, na wyjściu: 12. Dla wygody możesz ograniczyć nazwy miesięcy do pierwszych 3 liter.
8. Korzystając z kodu dla zadania 4, zsumuj odwrotności kolejnych liczb pierwszych (zaczynając od 2), aż suma przekroczy 2. Wypisz tę sumę i liczbę zsumowanych składników.
9. Wczytaj liczbę całkowitą (być może ujemną), a następnie wypisz jej cyfry słownie. Przykład: 0 – „zero”, -147 – „minus jeden cztery siedem”.
10. Wprowadź 5 słów z klawiatury, a następnie wypisz słowo najdłuższe (wskazówki: wrzuć je na listę, obejrzyj składnię funkcji max).
11. Wczytaj parametry równania kwadratowego, tj. $ax^2 + bx + c = 0$, zakładając że $a \neq 0$, a następnie oblicz pierwiastki (uwaga: mogą być zespolone!). Wskazówka: funkcja sqrt z modułu cmath (nie pomyl z math!) zwraca pierwiastek zespolony.
12. Ile jest różnych znaków w treści tego zadania?
13. Wylosuj 13 liczb z przedziału [1,50], bez powtórzeń, a następnie policz ich średnią. (Pomijając import, można to rozwiązać w jednej linii.)

Własne funkcje

Funkcja to podprogram, zwykle zawierający parametry (argumenty), który może coś zwrócić. Składnia jest bardzo prosta:

```
def dodaj(a,b):
  return a + b
```

Zauważmy, że taka funkcja zadziała np. gdy argumenty są typu *int*, wynikiem będzie ich suma, ale także gdy argumenty są stringami (wynikiem będzie konkatenacja stringów). Sprawdź:

```
x, y = 3, -10
print dodaj(x,y)
s1 = "Ala "
s2 = dodaj(s1, "zgubiła psa i płacze.")
print s2
```

W Pythonie – w przeciwieństwie do takich języków jak np. C++, Java, Pascal – funkcja może być argumentem jakieś funkcji, można być wynikiem (wartością zwracaną) funkcji, można ją podstawiać pod zmienne etc. Bardzo łatwo sprawdzić, co jest funkcją, a co nie: do tego służy wbudowana funkcja callable():

```
callable(list), callable(int), callable(len), callable(-2)
```

Wynikiem jest (True, True, True, False), gdyż list i int to nie tylko typy, ale i funkcje odpowiednich konwersji, len jest funkcją wbudowaną, a -2 oczywiście funkcją nie jest. Sprawdź samodzielnie, co zwróci callable dla wyżej zdefiniowanej funkcji *dodaj* oraz dla funkcji log z modułu math.
Co znaczy: podstawić funkcję pod zmienną? Ano, dzięki temu można nadać funkcjom (wbudowanym w język albo wziętym z jakichś „gotowych" modułów, albo naszym własnym) inne nazwy, aliasy:

```
import math
pierwiastek = math.sqrt   # bez nawiasów!
print pierwiastek(2)
print math.sqrt(2)  # oczywiście „po staremu" nadal można
długość = len
print długość("Jarzębina czerwona")
```

Tyle że działa to dla funkcji, a nie np. dla słów kluczowych, dlatego ambitna próba stworzenia „polskiego Pythona" zakończy się szybką klapą:

```
dla = for
SyntaxError: invalid syntax
```

Czy w funkcji możemy zmienić przekazane argumenty, tj. czy będą zmiany widoczne na zewnątrz funkcji? Zależy od typu argumentu: ponieważ stringi, krotki i liczby są niemutowalne, to one nie zostaną zmienione. Listy – mogą być.

```
def bzdury(i, j):
  i += 2
  j *= 2

a, b = 5, "abc"
bzdury(a, b)
print a, b  # jak sądzisz, co zostanie wypisane?

def bzdury2(li):
  for i in range(len(li)):
  li[i] *= 2

moja_lista = [3, 10, "a"]
bzdury2(moja_lista)
print moja_lista
```

A gdyby operacja na elemencie listy w funkcji *bzdury2* wyglądała tak: li[i] -= 2 ? Wtedy otrzymalibyśmy TypeError, gdyż od stringu nie można odjąć liczby. Morał: uważaj! Typów argumentów funkcji się w Pythonie nie określa, ale to nie znaczy, że wszystko wszędzie „przejdzie" (mimo że interpreter nie zaprotestuje). Ot, taki urok języków dynamicznych.

Zauważ, że funkcja może coś zwracać (za pomocą instrukcji return), ale nie musi. Ten drugi przypadek kojarzy nam się z funkcjami void w językach C-podobnych (albo z procedurami w Pascalu / Delphi), ale w Pythonie różnic składniowych nie ma. Ot, nie pisze się w nagłówku, co funkcja ma zwrócić.

Funkcja może zwrócić wiele wartości (co de facto jest zwróceniem krotki), np.

```
def suma_roznica_iloczyn(x, y):
  return x+y, x-y, x*y
```

Zapoznaj się też z taką składnią, używaną gdy nie jest znana liczba argumentów:

```
def sumuj(powitanie, *ell):
# gwiazdka oznacza: bierz argumenty od tego miejsca do końca
  print powitanie
  s = 0
  for i in ell:
    s += i
  return s

print sumuj("napis powitalny :-)", 4, 2, 3, -1, 8)
print sumuj("jeszcze raz", 4, 1)
```

Inne ciekawe cechy funkcji w Pythonie to możliwość ustawiania wartości domyślnych wybranych argumentów oraz wywoływanie funkcji z użyciem nazw argumentów, wówczas w dowolnej kolejności. Niżej przykłady.

```
def cena_z_vatem(cena, vat=22):
  """składnia: cena, vat, oba parametry liczbowe, parametr cena w %"""
  return cena * (1 + float(vat)/100)

pendrive = 80
print "cena pendrive'a netto:", pendrive, "z VAT-em:", cena_z_vatem(pendrive)
gazeta = 1.869
print "cena gazety brutto:", cena_z_vatem(gazeta, 7)
```

W powyższym przykładzie zaopatrzyliśmy naszą funkcję w tzw. docstring (zaraz za wierszem nagłówkowym). Tekst ten ukaże się, jeśli napiszemy help(cena_z_vatem). Docstring może być wielolinijkowy. Warto w ten sposób dokumentować tworzony przez siebie kod. Inny sposób wyświetlenia docstringa to: cena_z_vatem.__doc__ (tekst jest ten sam, inny jest nieco forma wyświetlania, przy *help* nieco bardziej rozwlekła, choć może czytelniejsza). Obejrzyjmy sobie parę docstringów dla funkcji z biblioteki standardowej, np.

```
import math, random
help(math.modf)
math.log.__doc__
help(random.randint)
help(pierwiastek)  # o ile wcześniej zrobiliśmy przypisanie pierwiastek = math.sqrt
```

W ostatnim przykładzie wyświetla się docstring dla funkcji math.sqrt, a więc słowo *pierwiastek* w nim nie występuje. Cóż, trzeba się z tym pogodzić. Jak komuś mało, może zdefiniować swoją funkcję o nazwie *pierwiastek*, opatrzoną stosownym docstringiem (np. w języku polskim).

Wracamy do parametrów domyślnych i wymienianych z nazwy.

```
def szlaczek(symbol = "#", ile = 80):
  print symbol * ile

szlaczek(ile = 40)  # wypisze 40 haszy
szlaczek(ile = 20, symbol = "$")  # wypisze 20 dolarów
szlaczek()  # oba argumenty domyślne, czyli 80 znaków #
```

Ale oczywiście z nazw argumentów ani wartości domyślnych nie ma konieczności korzystać, i można pisać tak:
szlaczek("a", 15)

Albo korzystamy wybiórczo:
szlaczek("a")

Ale wywołanie szlaczek(40) jest błędne! Jeśli pomijamy jakieś argumenty (bo mają wartości domyślne) i do tego nie używamy ich nazw, to trzeba po kolei.

Zadania:

14. Napisz funkcję zwracającą wartość logiczną (True albo False) w zależności od tego, czy przekazany argument typu string jest palindromem (przykłady palindromów: kajak, madam), czy nie. Możesz dodać linię: if type(s) != str: return False (o ile argument funkcji oznaczony jest przez s).
15. Na podstawie rozwiązania zadania 4 napisz funkcję zwracającą listę liczb pierwszych z przedziału domkniętego [a, b], gdzie a i b są parametrami funkcji.
16. Napisz funkcję o nagłówku def filterdict(itemfunc, dictionary), zwracająca podsłownik słownika dictionary zawierający wyłącznie te pary, których klucze spełniają predykat (tj. funkcję zwracającą wartość bool) itemfunc. Przykład zastosowania: z książki telefonicznej zawierającej pary typu („Kowalski Jan”, 4123341) chcemy wyciągnąć użytkowników o nazwiskach na literę T (i stworzyć z nich osobny słownik).
17. Napisz funkcję, która dla argumentu typu string zwróci napis, z którego usunięto wszystkie spacje, a małe litery zostały zamienione na wielkie i vice versa (Przykład: "Ala ma lamę." → "aLAMALAMĘ.").
18. Napisz jednolinijkową funkcję rysującą prostokąt ze znaczków dolara o wymiarach a i b; domyślna liczba wierszy, tj. b, niech wynosi 1.
19. Napisz funkcję o zmiennej liczbie parametrów, będącą ocenami z zestawu 2,3,4,5, zwracającą średnią przekazanych ocen. Jeśli lista parametrów jest pusta, to funkcja ma zwrócić -1. Przykład użycia:
    print srednia_ocen(), srednia_ocen(3, 2), srednia_ocen(5, 4, 5, 3, 3) # -1 2.5 4.0

Pożyteczne narzędzia: split, join, sum, map, filter, reduce

Nie omówiliśmy jeszcze dwóch niezwykle przydatnych metod działających dla obiektów typu string. Metoda split() tworzy listę słów danego napisu, metoda join() tworzy string łącząc stringi na liście przekazanej jako argument.

Przykłady:

```
s = "Pies, lis i 2 koty"
lista = s.split()
print lista
s1 = "*".join(lista)
print s1
```

Wypróbuj join na rzecz stringu "***". Albo na rzecz stringu pustego.

Wnikliwy obserwator dostrzeże, że tokenizacja nie jest może doskonała: pierwsze słowo listy w naszym przykładzie kończy się przecinkiem, co może być niepożądane przy tworzeniu rozmaitych słowników, list użytkowników *etc*. Brak argumentu przy split() oznacza (dowolnej długości) ciąg białych znaków, ale gdybyśmy mieli inne separatory, np. dla:

```
s = "Pies,lis,kot"
```

to wystarczyłoby utworzyć listę słów w taki sposób:

```
lista = s.split(",")
```

Niestety, nasz przypadek jest jeszcze inny (separatorem jest albo spacja, albo np. przecinek+spacja) i rozwiązanie „problemu" odłożymy na później, gdy będziemy znali wyrażenia regularne, gdyż one są tu najprostszą drogą do osiągnięcia celu.

Oto jeszcze parę przykładów użycia split lub join:

```
email = "Lord.Vader@dark.side.pl"
lista = email.split("@")
page_footer = "Pisz do mnie pod: " + lista[0] + " (małpa) " + lista[1]
print page_footer
```

A może ambitniej chcemy się chronić przed spamem? Proponuję taki kodzik (po dwóch pierwszych liniach powyższego):

```
lista2 = []
for i in lista:
  lista2.append(i.split("."))
page_footer = "Pisz do mnie pod: " + " (kropka) ".join(lista2[0]) + " (małpa) " + \
  " (kropka) ".join(lista2[1])
print page_footer
```

```
fraza = "to be or not to be -- that is the question"
print "Napis", "\"" + fraza + "\"", "zawiera", len(set(fraza.split())), "różnych słów."
```

```
liczby = ["3", "4", "1", "10"]
print "+".join(liczby)
```

Teraz funkcja sum() – bardzo prosta i intuicyjna, zwraca sumę liczb w przekazanej sekwencji (liście lub krotce). Przykłady:

```
sum([1, 3.5, -2])
```

```
sum((1, 3.5, -2))  # ale nie tak: sum(1, 3.5, -2)
sum(range(3, 6))
sum([3+2j, 1+1j, 5])
```

Oczywiście sum **nie** służy do konkatenacji stringów na liście (od tego przecież jest join()).

Funkcja wbudowana map(f, sekw) przykłada funkcję f do kolejnych elementów sekwencji sekw i zwraca nową sekwencję o tej samej długości co sekw. Prosty przykład:

```
lista = map(len, ["To", "są", "elementy", "listy"])
print lista  # wyświetli [2, 2, 8, 5]
```

Powiedzmy, że chcemy mieć na liście pierwiastki kolejnych liczb całkowitych od 1 do 10. Piszemy:

```
import math
we = range(1, 11)
wy = map(math.sqrt, wy)  # obejrzyj wy
```

Można też krócej:

```
import math
wy = map(math, sqrt, range(1, 11))
```

Czy można użyć własnej funkcji? Oczywiście. Powiedzmy, że chcemy zrobić działanie odwrotne, tj. wyprodukować kwadraty elementów listy (np. nadal 1..10). Piszemy:

```
def kwadrat(x):
  return x*x

wy = map(kwadrat, range(1, 11))
```

Proste, ale trochę człowieka irytuje, że dla tak błahego celu trzeba było pisać osobną funkcję. Na szczęście wcale nie trzeba. Można wykorzystać tzw. funkcję anonimową – najpierw przykład:

```
wy = map(lambda x: x*x, range(1, 11))
```

Ogólnie w funkcjach anonimowych po słowie *lambda* podaje się argumenty funkcji, ale w tym przypadku, przy map, funkcja jest 1-argumentowa. Po dwukropku jest zwracane wyrażenie, czyli wszystko, o co nam chodzi.

Chcemy napisy na liście skrócić do pierwszych 2 liter – nic prostszego:

```
pory_roku = ["wiosna", "lato", "jesień", "zima"]
pory_skrót = map(lambda s: s[:2], pory_roku)
print pory_skrót
```

A może „rozstrzelić” napis, wstawiając 2 spacje po każdej literce?

```
s1 = "abrakadabra"
```

```
s2 = "".join(map(lambda ch: ch+"  ", s1))
print s2
```

Ostatni przykład był ciut bardziej skomplikowany: wyjściem *map* jest wszak lista, więc trzeba ją dopiero skleić w string – dlatego join(), a łącznikiem string pusty.

Niestety, ten przykład był z kategorii *overkill*: to samo można było osiągnąć prościej, wymieniając kluczową linię na
`s2 = "  ".join(list(s1))`

To samo? Hm, nie do końca – odpowiedz, dlaczego napisy s2 w obu przypadkach (nieco) się różnią.

Inny przykład: chcemy spacje w stringu s1 zastąpić znakami podkreślenia:
`s2 = "".join(map(lambda ch: "_" if ch==" " else ch, s1))`

Tu wszedł nowy element: wyrażenie trójkowe (dopiero od wersji 2.5 Pythona). Składnia: wartość1 if warunek else wartość2. Przy funkcjach anonimowych konstrukcja jak znalazł.

Częste wykorzystanie *map* to hurtowa konwersja liczb na napisy lub odwrotnie:

```
li_int = [4, 10, 18, 6, 200]
li_str = map(str, li_int)
for i in li_str:
  print i.rjust(6)
```

Powtórzmy, bo ważne: *map* zwraca nową sekwencję (dokładniej listę), nie modyfikuje starej.

Powiedzieliśmy, że map zachowuje długość sekwencji oryginalnej. Czasem jednak chcemy ją skrócić, odfiltrować. Innymi słowy, zostawić tylko te elementy, które spełniają jakieś kryterium. Mamy do tego w Pythonie funkcję *filter*:

```
print filter(lambda x: x>10, [2, 4, 20, 5, 10])  # na zwróconej liście jest tylko [20]

s = "Adam Mickiewicz jest autorem Dziadów i Świtezianki"
# wypisz słowa z wielkiej litery
li = filter(lambda w: w[0].isupper(), s.split())
print li  # możesz zmienić na ładniejszą postać wypisywania
```

Słownik zawiera pary: osoba, roczny dochód. Chcemy policzyć średnią dochodu tylko wśród osób, których dochód przekracza 100 tys. Nietrudne:

```
dane = { "Bjarne Stroustrup": 800000, "Harry Jones": 35000, "Jan Kowalski": 84000, \
  "John Silverberg": 120000 }
dochody = map(lambda i: dane[i], dane)  # i – klucz, dane[i] – skojarzona z nim wartość
odfiltrowane = filter(lambda forsa: forsa > 100000, dochody)
print float(sum(odfiltrowane)) / len(odfiltrowane)

pliki = ["dane.dat", "yeyeye.mp3", "winword.exe", "jazz01.MP3", "La Cucaracha.mp3"]
empetrójki = filter(lambda s: s.lower().endswith(".mp3"), pliki)
```

```
print empetrójki
```

Oczywiście, można też pisać trochę bardziej rozwlekle:

```
empetrójki = filter(lambda s: s.lower().endswith(".mp3") == True, pliki)
```

A zatem inne pliki otrzymamy poprzez:

```
pozostałe = filter(lambda s: s.lower().endswith(".mp3") == False, pliki)
```

albo też

```
pozostałe = filter(lambda s: not s.lower().endswith(".mp3"), pliki)
```

Funkcja *filter* zastosowana na stringu zwróci string (a nie listę, jak przy *map*). Podobnie z krotką.

I wreszcie *reduce*, stosunkowo rzadko używane, ale czasem wygodne. Tym razem przykładamy funkcję dwuargumentową: najpierw do dwóch pierwszych elementów sekwencji, potem pierwszym argumentem jest otrzymany wynik, a drugim – trzeci element sekwencji, itd. do końca sekwencji. Brzmi mętnie? No to przykłady:

```
silnia = reduce(lambda a, b: a*b, range(1, 7))  # 6!
binarne_OR = reduce(lambda a, b: a | b, [1, 4, 16, 3])
```

Powiedzmy, że (inspiracja wzięta z książki „Beginning Python" M. L. Hetlanda) studentowi I roku informatyki chcemy zademonstrować najbardziej podstawowe algorytmy i konstrukcje programistyczne. Jednym z klasycznych zadań jest szukanie maksimum w sekwencji (np. tablicy) liczb. Chcemy nie tylko dać końcową odpowiedź, ale pokazać krok po kroku, jak zmienia się ów max w trakcie szukania. Oto stosowny kod Pythona:

```
def max_z_podglądem(a, b):
  print "szukam większej liczby z pary", a, "oraz", b, "; jest nią", max(a, b)
  return max(a, b)  # tak, jest gotowa funkcja max w Pythonie, podobnie jak i min

print reduce(max_z_podglądem, [3, 1, 3, 9, 4, 11, 6, 12, 13, -2])
```

20. Dana jest lista liczb zespolonych. Możliwie prosto (1 linijka!) wyprodukuj listę modułów tych liczb, oczywiście w tej samej kolejności. Wskazówka: dla liczb zespolonych do części rzeczywistej i urojonej odwołujemy się przez pola real, imag.
21. Wczytaj od użytkownika listę 7 imion (każdy wpis zakończony Enterem), a następnie wypisz na ekran tylko te wpisy, które rzeczywiście mogą być imionami, to znaczy: pojedyńcze wyrazy, pisane z wielkiej litery, pozostałe litery małe. Wskazówka: metoda capitalize() dla stringów może się przydać.
22. Napisz funkcję o 1-linijkowym ciele zwracającą listę dzielników dodatnich argumentu będącego liczbą dodatnią. Wariant nieco trudniejszy (choć niekoniecznie w jednej linii): argument wejściowy może też być liczbą ujemną (zakładamy, że ma wtedy tylko dzielniki ujemne).
23. Napisz funkcję sprawdzającą, czy dana liczba dodatnia jest liczbą doskonałą (tj. jest sumą swych dzielników właściwych, np. 6 = 1 + 2 + 3). Przetestuj tę funkcję na wszystkich liczbach od 1 do 10000.

24. Dany jest napis s = "to jest 14-ta próba napisania tego programu, a miały być tylko 4 !!!". Zwróć konkatenację samych cyfr z tego napisu, w kolejności. Wskazówka: metoda s.isdigit().

Pliki

Pamięć RAM jest ulotna; żeby przechowywać dane w sposób trwały, należy wykorzystać pamięć dyskową (albo np. Flash). Typowym trwałym „pojemnikiem" na dane w pamięci zewnętrznej jest plik. Parę przykładów.

```
f = open("c:/topsecret/mypassword.txt", "rt")
text = f.read()
```

Zmienna text przechowuje zawartość wskazanego pliku, jako napis (string). Co więcej, gdyby f został otwarty w trybie binarnym ('b' zamiast 't'), to też funkcja read() zwróciłaby string.

Obiekty plikowe mają 3 atrybuty, których znaczenia domyślisz się po przykładzie:

```
print f.name, f.mode, f.closed
```

Wykonaj teraz operację f.close() i wyświetl atrybut closed tego pliku.

Jeżeli chcemy plik „na raz" wczytać do pamięci (żeby analizować jego zawartość, natomiast pliku nie zmieniać), nie trzeba nawet tworzyć zmiennej plikowej. Ściągnij książkę „Oliver Twist" Dickensa spod http://www.gutenberg.org/etext/730 (w formacie txt, tj. „plain text") i zachowaj np. w katalogu głównym dysku C. Następnie wykonaj:

```
words = open("c:/730.txt").read().split()
print words[:30]
words.count("Oliver")
words.count("Twist")
```

Jak widzisz, wszystkie słowa książki „siedzą" na liście words i możemy prosto np. policzyć, ile razy w tekście występuje imię oraz nazwisko tytułowego bohatera. A może chcemy znać pozycje słowa „Twist" na liście słów? Oto kod:

```
for j, i in enumerate(words):
        if i == "Twist":
                print j,
```

Funkcja *enumerate*(lista), formalnie rzecz biorąc, zwraca obiekt enumeracji (wyliczenia), ale możemy o tym na razie zapomnieć. Jej użycie sprowadza się do odczytania bieżącego elementu listy **wraz z indeksem**. Sprawdź na mniejszym przykładzie:

```
li = [3, -5, "abc", 10]
for j, in in enumerate(li):
        print i, j
```

Wypróbuj też: list(enumerate(li)).

Ile różnych słów jest w naszym pliku? Nie musimy go ponownie czytać, bo przecież nie zmienialiśmy listy *words*. Piszemy po prostu:

```
print len(set(words)).
```

Nieraz wygodniej jest działać na poziomie całych wierszy. W takim przypadku użyjmy metody *readlines*(). Wczytajmy jeszcze raz cały plik, tym razem do listy wierszy i wypiszmy na ekran wiersze od 100-go do 120-ego. Żeby układ tekstu był „ładny" (wiersz w jednej linii), użyj pętli *for* oraz *print* w jej wnętrzu. A może widzisz co drugą linię pustą? Co zrobiłeś źle? Popatrz na linię print-a.

Zapis do pliku to oczywiście metoda *write* (argumentem jest string).

```
g = open("c:/output.txt", "w")
g.write("Oto kilka początkowych liczb pierwszych:\n")
for i in [2, 3, 5, 7, 11]:
  g.write(str(i)+" ")
g.close()
```

Można też jednym wywołaniem metody *writelines* zapisać do pliku listę stringów:

```
lista = ["napis1 ", "napis 2 ", "napis 3 "]
g.writelines(lista)
```

25. Wypisz 10 pierwszych linii danego pliku tekstowego – wspak.
26. Wypisz 10 najczęstszych słów w pliku tekstowym (np. „Oliverze Twiście") wraz z liczbą wystąpień. Wskazówka: użyj słownika.
27. Wypisz na ekran tylko te linie pliku tekstowego, które zawierają przynajmniej 10 liter 'a' (dla O.T. jest takich linii 7, pierwsza to „him a thousand embraces, and what Oliver wanted a great deal more, a"). W zasadzie możesz to zrobić w jednej linii (wskazówka: filter).
28. Zapisz do pliku tekstowego tabliczkę mnożenia 10 x 10 (p. zadanie 3), zadbaj o wyrównanie liczb (metoda *rjust* dla stringów).

Wykorzystanie systemu operacyjnego

Nieraz (np. w praktyce administratora systemu sieciowego) zachodzi potrzeba wyszukania plików o określonych nazwach czy atrybutach, skasowania za pomocą skryptu jakichś plików, zsumowania miejsca dyskowego zajmowanego przez dane drzewo katalogów etc. Biblioteka standardowa Pythona zawiera narzędzia umożliwiające wykonywanie tego typu działań, bez względu na używany system operacyjny.

Zaimportujmy na początek moduł os:

```
import os
```

Mamy tam m.in. funkcje:

getcwd() – zwraca (jako string) bieżący katalog

chdir(*nowy_katalog*) – nietrudno odgadnąć... Wypróbuj z podawaniem ścieżki względnej i bezwzględnej

mkdir(*nowy_katalog*) – tworzy katalog

listdir(*katalog*), np. listdir("."), czyli zawartość katalogu bieżącego, jako lista stringów

Użycie ostatniej z ww. funkcji jest szczególnie wygodne, jeśli możemy odfiltrować nazwy plików. Można do tego użyć standardowych funkcji dla stringów albo wyrażeń regularnych, ale w typowych zastosowaniach zwykle wygodniejszy jest moduł fnmatch („fn" niech się kojarzy z „filename"):

```
import fnmatch
print fnmatch.fnmatch("song01.mp3", "*.mp3")  # (sciezka / nazwa pliku, wzorzec)
```

Wypisuje True.

Można użyć też klas znaków, np.

```
print fnmatch.fnmatch("song01.mp3", "*[023].mp3")
```

Teraz False.

Napisz skrypt (może być 1 linia) wypisujący wszystkie pliki z katalogu Lib zainstalowanego Pythona o nazwach zaczynających się na 's' i rozszerzeniu .py.

Mamy też do dyspozycji moduł os.path, który może wyglądać na „podmoduł" modułu os, ale naprawdę jest referencją do osobnego modułu zależnego od platformy, jakiej używamy (np. w systemach Windows jest to moduł ntpath, ale zaleca się importowanie os.path, ze względu na przenośność kodu).

Dużo rzeczy jest w os.path, np.:

```
os.path.split("c:/Python26/Lib/compiler")
os.path.exists("c:/Python26/Lib/1")
os.path.isfile("c:/Python26/Lib")  # zwróci False
os.path.isdir("c:/Python26/Lib")  # zwróci True
os.path.getsize("c:/Python26/Lib/atexit.py")  # rozmiar w bajtach, np. 1770L
os.path.getsize("c:/Python26/Lib")  # dla katalogów getsize zwraca 0
```

29. Napisz program, który zsumuje wielkość wszystkich plików w zadanym katalogu, przy czym nie dbaj o ewent. podkatalogi. Uwaga: nie uwzględniaj wielkości klastra.
30. J.w., ale tym razem uwzględnij strukturę drzewiastą katalogu (jego podkatalogi też mogą mieć podkatalogi etc., więc narzuca się podejście rekurencyjne).
31. Znajdź najdłuższy plik w danym katalogu (z uwzględnieniem ewent. podkatalogów).
32. Zwróć liczbę unikalnych nazw plików (podkatalogów nie traktujemy jak pliki) w danym drzewie katalogów (oczywiście nazwy wraz ze ścieżką są unikalne, ale same nazwy plików niekoniecznie, np. readme.txt może występować wielokrotnie, w różnych podkatalogach).