Podręcznik programowania w języku Python

złożył: **Jacek Patka**

źródło: **http://uoo.univ.szczecin.pl/~jakubs/news/**

Spis treści:

LEKCJA 1 - WPROWADZENIE

1.1 Czym jest Python

"Python is a general-purpose open source computer programming language, optimized for quality, productivity, portability, and integration. It is used by hundreds of thousands of developers around the world, in areas such as Internet scripting, systems programming, user interfaces, product customization, and more. Among other things, Python sports object-oriented programming (OOP); a remarkably simple, readable, and maintainable syntax; integration with C components; and a vast collection of precoded interfaces and utilities."
- M.Lutz

Twórcą języka Python jest Guido van Rossum. Nazwa Python wywodzi się od tytułu serii programów satyrycznych emitowanych przez telewizję BBC w latach 70. ubiegłego wieku.

David Ascher i Mark Lutz już w roku 2003 oceniali światową liczbę użytkowników Pythona na 1 milion. Poza indywidualnymi osobami wymieniali takie znane kompanie jak Google, Yahoo!, Hewlett-Packard, Seagate, IBM i Industrial Light and Magic.

Python jest językiem interpretowanym, co w stosunku do języków kompilowanych takich jak C czy Pascal z jednej strony przekłada się na większą łatwość modyfikacji gotowego programu, z drugiej na większą powolność działania.
Rzeczywisty sposób wykonywania programu w Pythonie zbliżony jest do języka Java. Program źródłowy napisany w języku Python najpierw kompilowany jest do postaci pośredniej (byte-code), która następnie wykonywana jest przez Wirtualną Maszynę Pythona (PVM).

1.2 Skąd wziąć i jak zainstalować Pythona

Interpreter Pythona może być bezpłatnie pobrany ze strony http://www.python.org/
Klasyczny interpreter Pythona napisany został w języku C i dostępny jest w postaci skompilowanej dla szeregu systemów operacyjnych, w tym najpopularniejszych: Windows, Sony PlayStation 2 i różnych dystrybucji Linuxa.
Dostępna jest także odmiana Pythona napisana w języku Java, nazywana Jython.

Do nauki kursu zalecana jest wersja Pythona 2.6.
W celu zainstalowania Pythona w systemie Windows, należy otworzyć pobrany ze strony Pythona plik http://www.python.org/ftp/python/2.6.6/python-2.6.6.msi.
Pierwsze okno dialogowe, które pokaże się, pozwala wybrać instalację dla jednego lub wszystkich użytkowników systemu.
Następnie wybieramy katalog, w którym Python zostanie zainstalowany (zalecane domyślne "C:\Python26").
Kolejne okno pozwala wybrać składniki, które zamierzamy zainstalować (zalecane domyślne - wszystkie).
W tym momencie program zostanie zainstalowany.
Kliknięcie przycisku "Finish" zamyka program instalacyjny.

1.3 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!

Literatura w języku angielskim:

Michael Dawson: *Python Programming for the Absolute Beginner*. Premier Press 2003, ISBN 1-592-00073-8.
Mark Lutz: *Programming Python, 2nd Edition*. O'Reilly 2001, ISBN 0-596-00085-5.
Alex Martelli: *Python in a Nutshell*. O'Reilly 2003, ISBN 0-596-00188-6.
David Ascher, Mark Lutz: *Learning Python, 2nd Edition*. O'Reilly 2003, ISBN 0-596-00281-5.

1.4 Strony internetowe w języku polskim

Polskie tłumaczenie dokumentacji
http://www.python.org.pl/

Kody źródłowe
http://python.kofeina.net/

Kurs Pythona
http://www.mychkm.webpark.pl/python/

LEKCJA 2 - PODSTAWY PYTHONA

2.1 Uruchamianie Pythona w systemie Windows

Jeżeli Python został prawidłowo zainstalowany, aby uruchomić interpreter w trybie interaktywnym, klikamy przycisk "Start", dalej "Programy" ("Wszystkie Programy" w XP), wybieramy folder "Python 2.4", i na koniec "IDLE (Python GUI)". W oknie, które po chwili się pojawi, powinniśmy zobaczyć następujący tekst:

```
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.1
>>>
```

Radzę tak zmniejszyć wielkość okienka IDLE, by widzieć zawartość niniejszego dokumentu bez potrzeby przełączania się między okienkami.

2.2 Zalecana Konfiguracja Pythona

Przy pierwszym uruchomieniu Pythona, upewnijmy się że włączona jest funkcja automatycznego zapisu programów, oraz że IDLE koduje polskie znaki we właściwy sposób.
Z menu **Options** wybierzmy **Configure IDLE...**, kliknijmy zakładkę **General**, a następnie:

- ustawmy **Autosave Preference At Start of Run** (druga opcja od góry) na **No Prompt**
- ustawmy **Default Source Encoding** (przedostatnia opcja) na **Locale-defined**
- kliknijmy przycisk **OK**, aby zapamiętać zmiany.

2.3 Korzystanie z pomocy On-Line

Zanim przejdziemy dalej, powinniśmy wiedzieć, gdzie w razie czego szukać pomocy.
Poprzez kliknięcie na menu **Help** uzyskujemy dostęp do pomocy na temat środowiska programistycznego IDLE (**IDLE Help**) oraz pomocy na temat języka Python (**Python Docs F1**).
Pomoc na temat IDLE warto przejrzeć przy pierwszym z nim kontakcie, by zyskać ogólny ogląd o jego możliwościach. Na szukanie odpowiedzi na konkretne pytania, przyjdzie czas, gdy takowe się pojawią.
Podobnie z pomocą na temat Pythona: najważniejsze by wiedzieć gdzie szukać informacji. Strona główna dokumentacji Pythona zawiera odnośniki do poszczególnych jej sekcji. Szczególnie interesującą sekcją dla początkującego programisty Pythona jest "Tutorial", zawierający wprowadzenie do tego języka oraz opis najważniejszych jego konstrukcji. Opis wszystkich elementów języka Python znajdziemy w sekcji "Language Reference", natomiast opis funkcji dołączanych ze standardowych bibliotek w sekcji "Library Reference".

2.4 Tryb interaktywny Pythona

Widoczny po uruchomieniu trybu interaktywnego Pythona znak zachęty ">>>" oznacza gotowość interpretera do wykonywania naszych poleceń.
Aby sprawdzić czy interakcja rzeczywiście działa, wpiszmy "credits" i wciśnijmy klawisz Enter; po chwili powinniśmy zobaczyć:

```
>>> credits
Thanks to CWI, CNRI, BeOpen.com, Zope Corporation and a cast of thousands for
supporting Python development.  See www.python.org for more information.
>>>
```

2.5 Liczby i operatory arytmetyczne

Tryb interaktywny Pythona może być używany jako kalkulator. Wpiszmy:

```
>>> 2+2
```

aby dodać dwie dwójki. Powinniśmy otrzymać następujący wynik:

```
4
>>>
```

Poniżej wypróbujemy pozostałe trzy podstawowe operatory matematyczne - odejmowanie, mnożenie i dzielenie:

```
>>> 4-1
3
>>> 2*3
6
>>> 4/2
2
>>>
```

Próba dzielenia przez zero kończy się w Pythonie wyświetleniem komunikatu o błędzie:

```
>>> 2/0

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    2/0
ZeroDivisionError: integer division or modulo by zero
>>>
```

Dodatkowe dwa operatory arytmetyczne dostępne w Pythonie to potęgowanie:

```
>>> 3**2
9
>>>
```

oraz reszta z dzielenia:

```
>>> 8%3
2
>>>
```

Pojedyncze wyrażenie może zawierać wiele operatorów. Są one wykonywane w kolejności znanej z algebry:

```
>>> 2+4/2
4
>>> 25+2*-3**3+12/3
-25
>>>
```

W przypadku konieczności zmiany kolejności wykonywania operatorów używamy nawiasów okrągłych:

```
>>> (3+5)/2
4
>>> -((2+1)*(8-3))**2
-225
>>>
```

2.6 Systemy liczbowe

Liczby całkowite mogą być w Pythonie zapisywane w systemach pozycyjnych innych niż dziesiętny: ósemkowym i szesnastkowym.
Liczbę w systemie ósemkowym zapisujemy poprzedzając jej wartość znakiem zera (interpreter Pythona zwraca tą samą wartość zapisaną dziesiętnie):

```
>>> 011
9
>>> 0764
500
>>>
```

Liczbę w systemie szesnastkowym zapisujemy poprzedzając jej wartość dwuznakiem "0x":

```
>>> 0X100
256
>>> 0xabc
2748
>>> 0xBADF00D
195948557
>>>
```

2.7 Typy liczb

Obok liczb całkowitych w Pythonie są dostępne trzy inne typy liczbowe:

- długie liczby całkowite,
- liczby rzeczywiste,
- liczby zespolone.

2.8 Długie liczby całkowite

Liczby całkowite mają rozmiar 32 bitów, stąd największą wartością jaką mogą przyjąć jest 2147483647. Większe wartości zapamiętywane są jako długie liczby całkowite, rozpoznajemy je po umieszczonej na końcu literze "L":

```
>>> 99999999999999999
99999999999999999L
>>>
```

Wartość mniejszą niż 2147483647 możemy oznaczyć jako długą sami dostawiając literę "L":

```
>>> 7l
7L
>>> 0xDEADF00L
233496320L
>>>
```

Jeżeli w wyrażeniu występuje choć jedna długa liczba całkowita, również rezultat jest długą liczbą całkowitą:

```
>>> 2+4*5L
22L
>>>
```

2.9 Liczby rzeczywiste

Liczby rzeczywiste mogą być w Pythonie przedstawione w formie ułamka dziesiętnego (z separatorem w postaci kropki, nie przecinka!):

```
>>> 2.5
2.5
>>>
```

lub w notacji naukowej (mantysa"E"+-wykładnik):

```
>>> 1e+3
1000.0
>>>
```

Wartość całkowitą możemy oznaczyć jako liczbę rzeczywistą dostawiając na końcu ".0". Porównajmy:

```
>>> 3/2
1
>>> 3.0/2.0
1.5
>>>
```

Aby odrzucić z wyniku dzielenia część ułamkową należy użyć podwójnego znaku dzielenia:

```
>>> 3.0//2.0
1.0
>>>
```

Jak widać, jeżeli w wyrażeniu występuje choć jedna liczba rzeczywista, również rezultat jest liczbą rzeczywistą. Jeszcze jeden przykład na to:

```
>>> 1.5e+1+5000000000L*3
15000000015.0
>>>
```

2.10 Liczby zespolone

Liczby zespolone są w Pythonie zapisywane jako suma części rzeczywistej i części urojonej (w celu prawidłowego wykonywania operacji arytmetycznych, najlepiej by suma była ujęta w nawiasy okrągłe); część urojona oznaczana jest przez dostawioną na jej końcu literę "J":

```
>>> -1+1j
(-1+1j)
>>> 0.5+0.5j
(0.5+0.5j)
>>>
```

Jeżeli w wyrażeniu występuje choć jedna liczba urojona, rezultat jest liczbą zespoloną:

```
>>> 1j**2
(-1+0j)
>>> 6.8*200L/(1+2j)
(272-544j)
>>>
```

2.11 Tworzenie i zmiana wartości zmiennych

Python pozwala używać zmiennych w celu przechowywania wartości dowolnego typu.
Utworzenie zmiennej polega na nadaniu jej początkowej wartości. Poniżej utworzymy zmienną a, nadając jej wartość 4:

```
>>> a=4
>>>
```

Sprawdźmy, `czy a rzeczywiście równe jest cztery:`

```
>>> a
4
>>>
```

Stwórzmy nową zmienną:

```
>>> b=5
>>>
```

Aby wykonać dwa polecenia Pythona w jednym wierszu rozdzielamy je średnikiem. Sprawdźmy wartości obu zmiennych:

```
>>> a;b
4
5
>>>
```

W każdej chwili możemy zmienić wartość zmiennej:

```
>>> a=7
>>> a
7
>>>
```

Do zmiennej możemy podstawić wartość innej zmiennej:

```
>>> a=b
>>> a;b
5
5
>>> b=3
>>> a;b
5
3
>>>
```

lub dowolnie skomplikowane wyrażenie:

```
>>> c=1+17*(32/8)
>>> d=(a*b)/(a+b+2.0)
>>> c;d
69
1.5
>>>
```

Jak widać, zmienna przyjmuje wartość typu rzeczywistego tylko wtedy, gdy wyrażenie, które do niej podstawiono było takiego typu (czyli zawierało przynajmniej jedną liczbę rzeczywistą - w powyższym przypadku 2.0).
Nowa wartość zmiennej może być wyliczona o dotychczasową wartość jej samej:

```
>>> a=a+7
>>> b=b-3
>>> c=c*2
>>> d=d/3
>>> a;b;c;d
12
0
138
0.5
>>>
```

Operację zmiany wartości zmiennej z wykorzystaniem jej dotychczasowej wartości można zapisać prościej, łącząc odpowiedni operator arytmetyczny ze znakiem równości:

```
>>> a+=2
>>> b-=2
>>> c*=2
>>> d/=2
>>> a;b;c;d
14
-2
276
0.25
>>>
```

Próba użycia zmiennej, której wcześniej nie nadano żadnej wartości, kończy się wyświetleniem komunikatu o błędzie:

```
>>> e

Traceback (most recent call last):
  File "<pyshell#30>", line 1, in -toplevel-
    e
NameError: name 'e' is not defined
>>>
```

Istniejącą zmienną można skasować instrukcją "del":

```
>>> del a
>>> a

Traceback (most recent call last):
  File "<pyshell#32>", line 1, in -toplevel-
    a
NameError: name 'a' is not defined
>>>
```

a potem stworzyć od nowa:

```
>>> a=71.0
>>> a
71.0
>>>
```

2.12 Zasady nazywania zmiennych

Nazwy zmiennych w Pythonie mogą być dowolnie długie:

```
>>> NieprawdopodobnieOkrutnieDlugaNazwaZupelnieNiewaznejZmiennej=1234
>>> NieprawdopodobnieOkrutnieDlugaNazwaZupelnieNiewaznejZmiennej*2
2468
>>>
```

i mogą zawierać zarówno małe, jak i duże litery alfabetu, tak łacińskiego, jak i polskiego. Takie same nazwy, ale napisane małymi bądź dużymi literami, oznaczają różne zmienne:

```
>>> aa=1
>>> Aa=2
>>> aA=3
>>> AA=4
>>> aa;Aa;aA;AA
1
2
3
4
>>>
```

Zmiennym nie można nadawać nazw zastrzeżonych dla instrukcji języka Python:

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

Próba użycia któregoś z powyższych słów kluczowych jako nazwy zmiennej, kończy się wyświetleniem komunikatu o błędzie:

```
>>> del=1
SyntaxError: invalid syntax
>>>
```

Nazwy nie mogą zawierać spacji:

```
>>> dwie nazwy=1
SyntaxError: invalid syntax
>>>
```

Mogą jednak zawierać znak podkreślenia:

```
>>> jedna_nazwa=2
>>> jedna_nazwa
2
>>>
```

Nazwy zmiennych mogą także zawierać cyfry:

```
>>> odpowiedz_nr_234=7
>>> odpowiedz_nr_234
7
>>>
```

Cyfry jednak nie mogą rozpoczynać nazwy zmiennej:

```
>>> 7mil=12
SyntaxError: invalid syntax
>>>
```

Znak podkreślenia może rozpoczynać nazwę zmiennej:

```
>>> _a=1
>>> _a
1
>>>
```

Jednak nie powinno się go w tym miejscu używać bez przemyślenia, gdyż w języku Python nazwy zaczynające się od "_" mają specjalne znaczenie - patrz dokumentacja punkty 6.12, 3.3 i 5.2.1. Ponadto, sam znak podkreślenia zwraca w trybie interaktywnym (tylko w tym trybie!) wartość ostatniego obliczonego wyrażenia:

```
>>> 7+3*2
13
>>> _
13
>>>
```

2.13 Napisy

Innym niż liczby rodzajem danych, którymi możemy posługiwać się w Pythonie są napisy.
Napisy ograniczamy cudzysłowami bądź apostrofami.
Jeżeli początek napisu oznaczyliśmy cudzysłowem, to tak samo powinniśmy go zakończyć:

```
>>> "napis"
'napis'
>>>
```

Jeżeli początek napisu oznaczyliśmy apostrofem, to tak samo powinniśmy go zakończyć:

```
>>> 'inny napis'
'inny napis'
>>>
```

```
W napisach ograniczonych cudzysłowami możemy używać apostrofów:
```

```
>>> "I can't help"
"I can't help"
>>>
```

```
W napisach ograniczonych apostrofami możemy używać cudzysłowów:
```

```
>>> '"Moby Dick" is thick'
'"Moby Dick" is thick'
>>>
```

Napisy ograniczone pojedynczymi cudzysłowami bądź apostrofami muszą kończyć się przed końcem linii:

```
>>> "napis bez końca
SyntaxError: EOL while scanning single-quoted string
>>>
```

Napisy mogą ciągnąć się przez wiele linii jeżeli ograniczymy je potrójnymi cudzysłowami:

```
>>> """Ten napis
ma
wiele
linii"""
'Ten napis\nma\nwiele\nlinii'
>>>
```

2.14 Podstawowe operacje na napisach

Napisy możemy podstawiać do zmiennych tak samo jak liczby:

```
>>> p="pies"
>>> k="kot"
>>> p
'pies'
>>> k
'kot'
>>>
```

Na napisach możemy wykonywać dwie podstawowe operacje - łączenie:

```
>>> p+k
'pieskot'
>>>
```

i powielanie:

```
>>> p*3
'piespiespies'
>>>
```

możemy je również łączyć:

```
>>> 2*k+" "+p
'kotkot pies'
>>>
```

2.15 Konwersja liczb na napisy

Aby skonwertować liczbę na napis posługujemy się odwróconym apostrofem (klawisz nad tabulatorem):

```
>>> a=2
>>> b=5
>>> "A="+`a`
'A=2'
>>> "B="+`b`
'B=5'
>>> `a`+`b`
'25'
>>>
```

2.16 Konwersja napisów na liczby

Jeżeli liczby przechowywane są w postaci napisów:

```
>>> x="1"
>>> y="2"
>>> x;y
'1'
'2'
>>>
```

to operacje na nich będą działały tak jak na napisach, a nie jak na liczbach:

```
>>> x+y
'12'
>>>
```

Aby móc wykonać na nich operacje arytmetyczne należy najpierw skonwertować napisy na liczby. Służy do tego jedna z czterech funkcji:

```
>>> int(x)
1
>>> long(x)
1L
>>> float(x)
1.0
>>> complex(x)
(1+0j)
>>>
```

A zatem

```
>>> int(x)+int(y)
3
>>>
```

2.17 Ćwiczenia kontrolne

I. Wiedząc, że pierwiastek n-tego stopnia z x równa się x do potęgi 1/n i wykorzystując wiedzę o użyciu liczb zespolonych w Pythonie, wylicz wartość pierwiastka drugiego stopnia z liczby -17.

II. Używając instrukcji Pythona oblicz resztę z dzielenia 17 przez 7 i zapamiętaj wynik w zmiennej o nazwie Z. Następnie, pojedynczym poleceniem Pythona i bez użycia nawiasów, przemnóż zmienną Z przez Z+3.

III. Spowoduj pojedynczym poleceniem Pythona, by na ekranie 20-krotnie wyświetliła się wartość wyrażenia 1.2e+3+34.5 każdorazowo rozdzielona średnikiem.

LEKCJA 3 - PIERWSZY PROGRAM

3.1 Tworzenie nowego programu

Po uruchomieniu IDLE'a zgłasza się tryb interaktywny Pythona. Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.**
Otworzy się nowe okno, przeznaczone do edycji programu. Zaczniemy od nadaia proramowi nazwy. W tym celu wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **prog1.py.** Rozszerzenie nazwy ".py" oznacza program źródłowy w języku Python (rozszerzenie ".pyc" oznacza zaś program skompilowany do postaci pośredniej). Klikamy na **Zapisz.**

3.2 Komentarze

Plik programu w języku Python zawiera ciąg poleceń do wykonania przez interpreter, zapisanych zgodnie ze składnią języka. Obok poleceń mogą znajdować się w nim również komentarze, opisujące cel lub sposób działania programu lub jego części. Komentarze w języku Python poprzedzamy znakiem **#** (sharp).
Zacznijmy nasz program od komentarza - napiszmy:

```
# To jest mój pierwszy program w języku Python
```

i wciśnijmy klawisz Enter. IDLE automatycznie oznacza komentarze kolorem czerwonym.
Komentarze mogą występować w dowolnej linii programu w Pythonie: cokolwiek pojawia się za znakiem # traktowane jest jako komentarz, aż do końca linii.
W kolejnej linii wpiszmy:

```
"Witaj świecie!"
```

i wciśnijmy klawisz Enter. IDLE automatycznie oznacza napisy kolorem zielonym.

3.3 Uruchamianie programu

W celu uruchomienia programu wybieramy z menu **Run** polecenie **Run Module** (lub wciskamy klawisz **F5**).
Po chwili nasz program zostanie uruchomiony. Efekt jego działania pojawi się w oknie trybu interaktywnego, a wygląda następująco:

```
>>> ================================ RESTART ====================
>>>
>>>
```

W trybie interaktywnym interpreter Pythona zwraca na ekran rezultat każdego przetwarzanego wyrażenia. Kiedy uruchomiony jest program, tak się jednak nie dzieje. Dlatego, mimo iż w drugiej linii programu znajduje się napis "Witaj świecie!", w oknie trybu interaktywnego nie został on wyświetlony.

3.4 Wyświetlanie danych

Aby wyświetlać napisy dowolnej treści z programu w Pythonie, należy posłużyć się instrukcją **print.**
Przejdźmy do okna z naszym programem "prog1.py" i dopiszmy **print** przed napisem "Witaj świecie!".
Otrzymamy:

```
# To jest mój pierwszy program w języku Python
print "Witaj świecie!"
```

Słowo "print" należy do nazw zastrzeżonych języka Python i wyświetlane jest przez IDLE'a na pomarańczowo.
Uruchommy program klawiszem **F5.** W oknie trybu interaktywnego powinniśmy zobaczyć:

```
>>> ================================ RESTART ====================
>>>
Witaj świecie!
```

```
>>>
```

3.5 Znaki sterujące

Tekst przeznaczony do wyświetlenia może zawierać specjalne znaki sterujące. Jednym z takich znaków jest tabulator zapisywany kombinacją znaków **\t**. Przejdźmy do okna z programem "prog1.py" i zmieńmy linię

```
print "Witaj świecie!"
```

na:

```
print "\tWitaj świecie!"
```

Po uruchomieniu programu klawiszem **F5,** w oknie trybu interaktywnego powinniśmy zobaczyć:

```
>>> ================================ RESTART ================================
>>>
                                Witaj świecie!
>>>
```

Innym znakiem sterującym jest znak końca linii oznaczany **\n**. Przejdźmy do okna z programem "prog1.py" i zmieńmy linię

```
print "\tWitaj świecie!"
```

na:

```
print "\tWitaj\nświecie!"
```

Po uruchomieniu programu klawiszem **F5,** w oknie trybu interaktywnego powinniśmy zobaczyć:

```
>>> ================================ RESTART ================================
>>>
                                    Witaj
świecie!
>>>
```

3.6 Wprowadzanie danych

Wiemy już, jak program może cokolwiek wyświetlać użytkownikowi. Teraz dowiemy się, jak wprowadzać dane od użytkownika do programu.
Przejdźmy do okna z programem "prog1.py" i zapiszmy go pod nową nazwą "prog2.py" wybierając z menu **File** polecenie **Save As.**
Zmieńmy pierwszą linię programu na:

```
# To już mój drugi program w języku Python
```

Następnie usuńmy drugą linię programu (print "\tWitaj\nświecie!\a").
Do wprowadzania danych od użytkownika służy wbudowana funkcja **raw_input**.
Dopiszmy więc do programu następujące dwie linie:

```
imie = raw_input("Jak masz na imię?")
print  "Witaj",imie,"!"
```

Po uruchomieniu programu klawiszem **F5,** w oknie trybu interaktywnego powinniśmy zobaczyć:

```
>>> ================================ RESTART ================================
>>>
Jak masz na imię?
```

W tym momencie program oczekuje na podanie przez nas swojego imienia (i potwierdzenia go Enterem). Jeżeli je tu wpiszemy, zobaczymy ciąg dalszy:

```
Jak masz na imię?Adam
Witaj Adam !
>>>
```

3.7 Program na obliczenie sumy dwóch liczb

Przejdźmy do okna z programem "prog2.py" i zapiszmy go pod nową nazwą "suma.py" wybierając z menu **File** polecenie **Save As.**
Usuńmy wszystkie linie programu wciskając kombinację klawiszy **CTRL**+**A**, a następnie klawisz **DELETE**.
Spróbujmy teraz napisać samodzielnie program, który wczyta od użytkownika dwie liczby całkowite, a następnie wyświetli ich sumę.
Prawidłowe rozwiązanie znajduje się poniżej:

```
# Ten program liczy sumę dwóch liczb całkowitych
x = raw_input("Podaj pierwszą z dwóch liczb:")
y = raw_input("Podaj drugą z dwóch liczb:")
print  "Suma liczb",x,"i",y,"wynosi",int(x)+int(y)
```

Po uruchomieniu programu klawiszem **F5,** w oknie trybu interaktywnego powinniśmy zobaczyć:

```
>>> ================================ RESTART ================================
>>>
Podaj pierwszą z dwóch liczb:4
Podaj drugą z dwóch liczb:3
Suma liczb 4 i 3 wynosi 7
>>>
```

Konieczność konwersji wprowadzanych danych (przy pomocy funkcji **int**), wynika stąd iż funkcja **raw_input** zwraca tekst wpisany przez użytkownika, a nie jego wartość liczbową.
Bez konwersji, program działał by nieprawidłowo:

```
>>> ================================ RESTART ================================
>>>
Podaj pierwszą z dwóch liczb:4
Podaj drugą z dwóch liczb:3
Suma liczb 4 i 3 wynosi 43
>>>
```

3.8 Ćwiczenia kontrolne

I. Napisz program "powiel.py", który wczyta od użytkownika pewien napis, a następnie wyświetli 30 kopii tego napisu, każda w osobnej linii.
II. Napisz program "pole_tr.py", który obliczy pole trójkąta, pod warunkiem że użytkownik poda wysokość i długość podstawy tego trójkąta. Uwzględnij, że wysokość i długość podstawy mogą być liczbami niecałkowitymi.
III. Napisz program "odsetki.py", który obliczy stan konta za N lat, gdzie stan początkowy konta wynosi SPK, a stopa oprocentowania P % rocznie (obowiązuje **miesięczna** kapitalizacja odsetek). N, SPK i P podaje użytkownik programu.

LEKCJA 4 - PISANIE ROZGAŁĘZIONYCH PROGRAMÓW

4.1 Operator równości

Po uruchomieniu IDLE'a zgłasza się tryb interaktywny Pythona. Wypróbujemy w nim działanie operatorów porównania.
Najważniejszym operatorem porównania jest operator równości.
Rezultatem każdego operatora porównania (w tym operatora równości) jest wartość typu logicznego, mogąca przybierać tylko dwie możliwe wartości: prawda lub fałsz.
Wpiszmy:

```
>>> 2==2
True
```

Uzyskany rezultat `True` oznacza, że prawdą jest, że 2 równe jest 2.

```
>>> 2==4
False
```

Uzyskany rezultat `False` oznacza, że nieprawdą jest, że 2 równe jest 4.

Poważnym błędem jest mylenie operator równości (= =) z instrukcją przypisania (=). Porównajmy:

```
>>> 2=2
SyntaxError: can't assign to literal
>>> a==2

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in -toplevel-
    a==2
NameError: name 'a' is not defined
>>> a=2
>>> a==2
True
>>>
```

4.2 Operatory nierówności

Do sprawdzania nierówności służy następujący operator:

```
>>> 2!=2
False
>>> 2!=4
True
```

Poniższy zapis również jest poprawny, lecz nie jest zalecany przez twórców Pythona:

```
>>> 2<>4
True
>>> 2<>2
False
>>>
```

Poniższy zapis jest niepoprawny:

```
>>> 2><4
SyntaxError: invalid syntax
```

Do sprawdzania ostrej większości i mniejszości służą następujące operatory:

```
>>> 2>4
False
>>> 2<4
True
>>> 2>1
True
>>> 2<1
False
```

Do sprawdzania nieostrej większości i mniejszości służą następujące operatory:

```
>>> 2>=4
False
>>> 2<=4
True
>>> 2<=2
True
>>> 2>=2
True
>>>
```

4.3 Porównania na napisach

W przypadku, gdy porównywane są napisy, kryterium porównania jest kolejność leksykograficzna:

```
>>> 'Ala'=='Ala'
True
>>> 'Ala'!='Ola'
True
>>> 'Ala'>='Ola'
False
>>> 'Ala'<'Ola'
True
>>> 'A'<'Aa'
True
>>> "123">"14"
False
```

4.4 Porównania na wyrażeniach

W przypadku, gdy porównywane są wyrażenia, kryterium porównania jest rezultat wyrażenia:

```
>>> 22.0/7>3.14
True
>>> a=14
>>> a**0.5<4**2
True
>>> (1+2+3+4)*0xA==0100+6*6
True
>>> 'Ala'*2=='Ala'+'Ala'
True
```

4.5 Porównania wielokrotne

Porównania mogą zawierać jednocześnie więcej niż jeden operator:

```
>>> 1<3<7
True
>>> 1>=1>2
False
>>> 2==2>1.1>-1>-1000
True
>>> 'Ala'<'Ola'<'Zenek'
True
```

4.6 Porównania łączone

Porównywać można także rezultaty porównań:

```
>>> (1==2)==(2==3)
True
```

Co należy rozumieć: Rezultat porównania 1 z 2 (fałsz) jest taki sam jak rezultat porównania 2 z 3 (również fałsz).
Więcej przykładów:

```
>>> (2>1)==(3>2)
True
>>> (2!=2)!=(3!=3)
False
```

4.7 Operator negacji

Operator negacji zaprzecza wynik porównania po nim:

```
>>> not 1==1
False
>>> not 2==0
True
```

4.8 Operator sumy logicznej

Operator sumy logicznej zwraca prawdę tylko wtedy, gdy choć jedno z połączonych nim wyrażeń jest prawdziwe:

```
>>> True or False
True
>>> False or not True
False
>>> 1==2 or 1==3 or 1==1
True
>>> 'Abc'=='Def' or 7>100 or 1==1
True
```

4.9 Operator iloczynu logicznego

Operator iloczynu logicznego zwraca prawdę tylko wtedy, gdy każde z połączonych nim wyrażeń jest prawdziwe:

```
>>> True and True
True
>>> True and False
False
>>> 1==1 and 2==3 or 3==3
True
>>> 1==1 and 2==3 and 3==3
False
```

4.10 Kolejność wykonywania operatorów

W pierwszej kolejności wykonywane są operatory porównania, później operator negacji, następnie iloczynu logicznego, a na końcu sumy logicznej (takie same operatory wykonywane są w kolejności od lewej do prawej). Porównajmy:

```
>>> False and False or True
True
>>> False and (False or True)
False
```

4.11 Instrukcja wyboru prostego IF

Aby uzależnić wykonanie instrukcji od rezultatu porównania, używamy instrukcji warunkowej (Uwaga: W trybie interaktywnym, na końcu każdej instrukcji złożonej - np. IF - konieczne jest dwukrotne wciśnięcie klawisza Enter):

```
>>> if 2==2: print "OK"

OK
```

Wyświetliło się "OK", bo warunek po IF jest spełniony.

```
>>> if 2!=2: print "OK"
```

Nie wyświetliło się "OK", bo warunek po IF nie został spełniony.

4.12 Instrukcja wyboru pełnego IF/ELSE

Można ustalić wykonanie różnych instrukcji dla obu możliwych przypadków:

```
>>> if 2!=2: print "Dziwne"
else: print "Normalne"

Normalne
```

4.13 Instrukcja wyboru wielokrotnego IF/ELIF/ELSE

Można ustalić wykonanie różnych instrukcji dla różnych przypadków:

```
>>> if 2==1: print "Dziwne 1"
elif 2==2: print "Normalne"
elif 2==3: print "Dziwne 3"
else: print "Dziwne inne"

Normalne
```

4.14 Tworzenie w programie rozgałęzień przy użyciu instrukcji wyboru

Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.**
Otworzy się nowe okno, przeznaczone do edycji programu. Zaczniemy od nadania programowi nazwy. W tym celu wybierzmy z menu **File** polecenie **Save As.** Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **warunek1.py**. Klikamy na **Zapisz**.

Przepiszmy następujący program:

```
# Pierwszy program rozgałęziony
print  "Ten program porównuje dwie liczby"
x = input ("Podaj pierwszą z dwóch liczb:")
y = input ("Podaj drugą z dwóch liczb:")
print  "Liczba",x,"w stosunku do",y,"jest:",       # Przecinek!
if x==y:
   print "taka sama"
elif x>y:
   print "większa"
else:
   print "mniejsza"
```

Uwagi do programu:

- Instrukcja **input** służy do wprowadzania liczb (przypomnijmy, że **raw_input** wprowadza tekst). Liczba może być podana przez użytkownika explicite, lub w postaci wyrażenia algebraicznego.
- Przecinek na końcu listy parametrów instrukcji **print** powoduje, że kolejna instrukcja **print** wyświetli tekst w tej samej (a nie następnej) linii

W celu uruchomienia programu wybieramy z menu **Run** polecenie **Run Module** (lub wciskamy klawisz **F5**).
Po chwili nasz program zostanie uruchomiony. Wprowadzamy dwie liczby i obserwujemy wynik. Np.:

```
Ten program porównuje dwie liczby
Podaj pierwszą z dwóch liczb:5
Podaj drugą z dwóch liczb:4.5
```

```
Liczba 5 w stosunku do 4.5 jest: większa
```

Lub z użyciem wyrażeń:

```
Ten program porównuje dwie liczby
Podaj pierwszą z dwóch liczb:6/2
Podaj drugą z dwóch liczb:1*3
Liczba 3 w stosunku do 3 jest: taka sama
```

4.15 Bloki warunkowe

Fragment programu, który wykonywany jest zależnie od wyniku instrukcji **if** określamy mianem bloku warunkowego. W języku Python blok warunkowy oznaczamy poprzez "wcięcie" tekstu o jeden tabulator w prawo.
Przyjrzyjmy się końcówce naszego programu:

```
else:
   print "mniejsza"
```

Załóżmy, że chcielibyśmy, aby na końcu programu wyświetlał się napis "dziękuję". Dopiszmy następującą linię:

```
else:
   print "mniejsza"
   print "Dziękuję!"
```

I uruchommy program. Jeżeli piszemy liczby 3 i 4, to napis "Dziękuję" zostanie wyświetlony:

```
Ten program porównuje dwie liczby
Podaj pierwszą z dwóch liczb:3
Podaj drugą z dwóch liczb:4
Liczba 3 w stosunku do 4 jest: mniejsza
Dziękuję!
```

Jeżeli jednak wpiszemy liczby 3 i 3, to napis "Dziękuję" nie zostanie wyświetlony:

```
Ten program porównuje dwie liczby
Podaj pierwszą z dwóch liczb:3
Podaj drugą z dwóch liczb:3
Liczba 3 w stosunku do 3 jest: taka sama
```

Dzieje się tak, gdyż instrukcja print "Dziękuję!" jest wewnątrz bloku warunkowego po "else:". Aby "Dziękuję" wyświetlane było niezależnie od wyniku porównania, zmieniamy program poprzez skasowanie tabulatora na początku ostatniej linii:

```
else:
   print "mniejsza"
print "Dziękuję!"
```

Teraz, po uruchomieniu programu, zobaczymy:

```
Ten program porównuje dwie liczby
Podaj pierwszą z dwóch liczb:3
Podaj drugą z dwóch liczb:3
Liczba 3 w stosunku do 3 jest: taka sama
Dziękuję!
```

Ważne: po instrukcji rozgałęziającej program (np. **if**, **elif**, **else**), wszystkie instrukcje należące do rozgałęzienia muszą być "wcięte" w prawo. Koniec gałęzi wyznacza pierwsza instrukcja, która jest na tym samym poziomie wysunięcia, co instrukcja rozgałęziająca.

4.16 Rozgałęzienia hierarchiczne

Poszczególne gałęzie programu można rozgałęziać na dalsze gałęzie.
Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.**
Otworzy się nowe okno, przeznaczone do edycji programu. Zaczniemy od nadania programowi nazwy. W tym celu wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **warunek2.py**. Klikamy na **Zapisz.**

Przepiszmy następujący program:

```
# Program wielokrotnie rozgałęziony
print  "Ten program przewidzi Twoją długość życia"
odp = raw_input ("Czy palisz papierosy?")
if odp == "Tak" or odp == "TAK" or odp == "tak" or odp == "no pewnie":
   odp = raw_input ("A czy się zaciągasz?")
   if odp == "Tak" or odp == "TAK" or odp == "tak" or odp == "no pewnie":
      print "Nie pożyjesz długo!"
   else:
      print "To pożyjesz długo!"
else:
   print "To pożyjesz bardzo długo!"
```

W celu uruchomienia programu wybieramy z menu **Run** polecenie **Run Module** (lub wciskamy klawisz **F5**). Po chwili nasz program zostanie uruchomiony. Wypróbujmy:

```
Ten program przewidzi Twoją długość życia
Czy palisz papierosy?no pewnie
A czy się zaciągasz?nie
To pożyjesz długo!
```

4.17 Ćwiczenia kontrolne

I. Napisz program "parzyste.py", który wczyta od użytkownika liczbę całkowitą i wyświetli informację, czy jest to liczba parzysta, czy nieparzysta.

II. Napisz program "calkowite.py", który wczyta od użytkownika liczbę i wyświetli informację, czy jest to liczba całkowita, czy niecałkowita.

III. Napisz program "prk.py", który obliczy wszystkie pierwiastki rzeczywiste równania kwadratowego o postaci $ax^2+bx+c=0$, gdzie a, b i c podaje użytkownik. Program powinien na początku sprawdzić, czy wprowadzone równanie jest rzeczywiście kwadratowe.

LEKCJA 5 - TYPY SEKWENCYJNE

5.1 Typy sekwencyjne

Po uruchomieniu IDLE'a zgłasza się tryb interaktywny Pythona.
Wykorzystamy go do poznania pythonowskich sekwencyjnych typów danych.
Sekwencyjne typy danych służą do zapamiętywania wielu wartości w pojedynczej zmiennej, w odróżnieniu od typów prostych, takich jak **int**, które w pojedynczej zmiennej mogą zachować tylko jedną wartość.

5.2 Typ napisowy

Do tej pory poznaliśmy już jeden typ sekwencyjny. Jest nim typ napisowy (string) - patrz punkty 2.13-2.16.
Przypomnijmy, że wartości napisów podajemy w cudzysłowach lub apostrofach:

```
>>> txt="napis"
>>> txt2='napis'
```

I, że możemy na nich wykonywać pewne operacje, np.:

```
>>> txt2+='owi nierówny'
>>> print txt2
napisowi nierówny
```

Napisy są sekwencjami znaków. Każdy typ sekwencyjny pozwala na dostęp do każdego swojego elementu z osobna. Aby uzyskać dostęp do znaku na określonej pozycji podajemy jej indeks (numer porządkowy liczony od lewej, zero oznacza pierwszy znak napisu) w nawiasach kwadratowych bezpośrednio po napisie:

```
>>> "abc"[0]
'a'
>>> "abc"[1]
'b'
>>> "abc"[2]
'c'
>>> txt[0]
'n'
>>> txt[1]
'a'
>>> txt[2]
'p'
>>> txt[3]
'i'
>>> txt[4]
's'
>>> txt[5]

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in -toplevel-
    txt[5]
IndexError: string index out of range
>>>
```

Powodem błędu była próba odczytania znaku o zbyt wysokim numerze, którego w napisie nie ma. Aby poznać długość napisu, posługujemy się funkcją len:

```
>>> len("abc")
3
>>> len(txt)
5
>>> len(txt2)
17
>>> len(txt*20)
100
>>>
```

Zliczać znaki możemy także od końca napisu w prawo. Używamy w tym celu indeksów ujemnych (-1 oznacza ostatni znak napisu):

```
>>> "abc"[-1]
'c'
>>> "abc"[-2]
'b'
>>> "abc"[-3]
'a'
>>> txt[-1]
's'
>>>
```

5.3 Kody ASCII

Pojedynczy znak zapisany jest jako liczba odpowiadająca kodowi ASCII określonego symbolu graficznego. Aby poznać kod ASCII określonego znaku, należy użyć funkcji **ord**:

```
>>> ord('A')
65
>>> ord('a')
97
>>> ord("1")
49
>>> ord("2")
50
```

Parametrem funkcji ord musi być pojedynczy znak, a nie wieloznakowy napis:

```
>>> ord(txt[0])
110
>>> ord(txt)

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in -toplevel-
    ord(txt)
TypeError: ord() expected a character, but string of length 5 found
>>>
```

Aby zamienić kod ASCII na odpowiadający mu znak, używamy funkcji **chr**:

```
>>> chr(65)
'A'
>>> chr(32)
' '
>>> chr(10)
'\n'
>>> chr(49)+chr(48)+chr(50)
'102'
>>>
```

5.4 Fragmenty napisu

Czasami interesuje nas nie pobranie z napisu pojedynczego znaku, ale wykrojenie ciągu znaków. Do wykrajania fragmentów napisów używamy zapisu z dwukropkiem:

Fragment napisu do ósmego znaku:

```
>>> txt2[:8]
'napisowi'
```

Fragment napisu od dziesiątego znaku:

```
>>> print txt2[9:]
```

```
nierówny
```

Fragment napisu od szóstego do dwunastego znaku:

```
>>> txt2[5:12]
'owi nie'
```

Co drugi znak z fragmentu napisu od szóstego do dwunastego znaku:

```
>>> txt2[5:12:2]
'oine'
```

Możemy także używać indeksów ujemnych:

```
>>> txt2[:-9]
'napisowi'
>>> txt2[-8:]
'nier\xf3wny'
>>> txt2[-12:-5]
'owi nie'
>>> txt2[-12:-5:2]
'oine'
```

5.5 Typ napisowy jako typ niezmienny

Sekwencyjne typy danych w Pythonie dzielimy na zmienne (mutable) i niezmienne (immutable). Typ napisowy należy do typów niezmiennych.
Typy niezmienne nie mogą zmieniać swojej wartości. A zatem zapis:

```
>>> txt+=chr(32)+txt2
>>> print txt
napis napisowi nierówny
```

tak naprawdę nie zmienia wartości zmiennej txt, ale tworzy nową zmienną o tej samej nazwie, a innej wartości, starą usuwając.
Konsekwencją niezmienności typu napisowego jest niemożność zmiany jego elementu:

```
>>> txt[2]='w'

Traceback (most recent call last):
  File "<pyshell#41>", line 1, in -toplevel-
    txt[2]='w'
TypeError: object does not support item assignment
>>>
```

ani fragmentu:

```
>>> txt[2:4]='wis'

Traceback (most recent call last):
  File "<pyshell#93>", line 1, in -toplevel-
    txt[2:4]='wis'
TypeError: object doesn't support slice assignment
>>>
```

Oczywiście, pożądaną operację możemy wykonać, inaczej formułując polecenie:

```
>>> txt2=txt[:2]+"w"+txt[3:]
>>> print txt2
nawis napisowi nierówny
>>>
```

5.6 Inne typy sekwencyjne

Napisy mogą przechowywać sekwencje znaków. Czasami potrzebujemy jednak zapamiętać sekwencje danych nie będących znakami, ale np. liczbami lub innymi sekwencjami.
Do przechowywania takich sekwencji służą w Pythonie: krotki (tuples) i listy (lists).
Elementy zarówno krotek, jak i list mogą być dowolnego typu.
Krotki są typem niezmiennym. Oznacza to, że mają z góry ustaloną długość, a zmiana wartości poszczególnych elementów z osobna nie jest możliwa.
Listy są typem zmiennym. Oznacza to, że można je łatwo skracać i wydłużać i jest możliwa zmiana wartości poszczególnych elementów z osobna.

5.7 Tworzenie i używanie list

Listy tworzymy używając nawiasów kwadratowych, rozdzielając ich elementy przecinkami.
Aby stworzyć listę trzech liczb naturalnych, napiszemy:

```
>>> lista1 = [1, 2, 3]
>>> lista1
[1, 2, 3]
```

Aby stworzyć listę złożoną z czterech napisów, napiszemy:

```
>>> lista2 = ["pierwszy", txt, txt2, "ostatni"]
>>> lista2
['pierwszy', 'napis napisowi nier\xf3wny', 'nawis napisowi nier\xf3wny', 'ostatni']
```

Listy mogą zawierać elementy różnych typów:

```
>>> lista3 = [1.0, 2, "trzy"]
>>> lista3
[1.0, 2, 'trzy']
```

Listy mogą zawierać inne listy:

```
>>> lista4=[ [1, 2, 3], ["Nocny", "Dzienny"] ]
>>> lista4
[[1, 2, 3], ['Nocny', 'Dzienny']]
```

Listy mogą być puste:

```
>>> lista_pusta = []
>>> lista_pusta
[]
>>> len(lista_pusta)
0
```

Listy mogą zawierać tylko jeden element:

```
>>> lista_jednoelementowa = [1]
>>> lista_jednoelementowa
[1]
```

Można odczytywać wybiórczo zawartość poszczególnych elementów listy:

```
>>> lista1[0]
1
>>> lista2[-1]
'ostatni'
```

Lub ich ciągów:

```
>>> lista1[1:]
[2, 3]
>>> lista2[0::3] # co trzeci element listy
```

```
['pierwszy', 'ostatni']
>>> lista4[1:]
[['Nocny', 'Dzienny']]
```

Listy można powielać:

```
>>> lista2*=2
>>> lista2
['pierwszy', 'napis napisowi nier\xf3wny', 'nawis napisowi nier\xf3wny', 'ostatni',
'pierwszy', 'napis napisowi nier\xf3wny', 'nawis napisowi nier\xf3wny', 'ostatni']
>>> []*1000
[]
>>> [1,2,3]*0
[]
```

Określać ich długość:

```
>>> len(lista2)
8
```

Skracać:

```
>>> lista2=lista2[:3]
>>> lista2
['pierwszy', 'napis napisowi nier\xf3wny', 'nawis napisowi nier\xf3wny']
```

Wydłużać:

```
>>> lista2+=['ostatni']
>>> lista2
['pierwszy', 'napis napisowi nier\xf3wny', 'nawis napisowi nier\xf3wny', 'ostatni']
```

5.8 Modyfikacja list

Listy są sekwencjami zmiennymi, można więc modyfikować ich fragmenty:

```
>>> lista3
[1.0, 2, 'trzy']
>>> lista3[0:2]=["jeden","dwa"]
>>> lista3
['jeden', 'dwa', 'trzy']
```

lub pojedyncze elementy:

```
>>> lista1
[1, 2, 3, 4]
>>> lista1[2]+=1
>>> lista1
[1, 2, 4, 4]
```

Można też usuwać elementy ze środka listy, tak samo jak w sekwencjach niezmiennych:

```
>>> lista2=lista2[:2]+lista2[3:]
>>> lista2
['pierwszy', 'napis napisowi nier\xf3wny', 'ostatni']
```

Lub prościej, z użyciem instrukcji **del,** np.:

```
>>> del lista2[1]
>>> lista2
['pierwszy', 'ostatni']
```

5.9 Porównywanie list

Porównywanie list odbywa się na zasadzie porównywania poszczególnych elementów:

- jeżeli elementy obu list są sobie równe, listy są równe
- jeżeli listy różnią się choć jednym elementem, to są nierówne
- jeżeli pierwszy element pierwszej listy jest większy od pierwszego elementu drugiej listy, to pierwsza lista jest większa od drugiej
- jeżeli pierwszy element pierwszej listy jest taki sam jak pierwszy element drugiej listy, decyduje porównanie drugich elementów, itd.
- element nieistniejący jest zawsze mniejszy od każdego innego elementu

```
>>> lista1 == [1, 2, 4, 4]
True
>>> lista1 != [1, 2, 3, 4]
True
>>> lista1 > [1, 2, 2, 5]
True
>>> lista1 < [1, 2, 4, 4, 5]
True
```

5.10 Sprawdzanie zawartości list

Aby sprawdzić, czy określona wartość znajduje się na liście, używamy operatora **in**:

```
>>> 1 in lista1
True
>>> 2 not in lista1
False
>>> "pierwszy" in lista2
True
```

5.11 Listy wielopoziomowe

W przypadku list wielopoziomowych, to jest takich, które zawierają inne listy, np.:

```
>>> lista4
[[1, 2, 3], ['Nocny', 'Dzienny']]
```

możliwy jest dostęp do poszczególnych elementów list podrzędnych poprzez użycie dwóch indeksów:

```
>>> lista4[1][0]
'Nocny'
>>> lista4[0][1]
2
```

Jako pierwszy podajemy zawsze indeks listy wyższego rzędu.

5.12 Typ listy jako typ zmienny

W Pythonie wszystkie sekwencje zmienne nie odnoszą się do określonych danych, ale do miejsca w pamięci, w którym te dane się znajdują.
W związku z tym przypisanie listy do listy nie kopiuje wartości, a jedynie wskaźnik do nich:

```
>>> lista5=lista1
>>> lista5
[1, 2, 4, 4]
```

A zatem od tej pory, niezależnie czy zmieniamy elementy listy 1 czy 5, zmieniamy obie, bo zmianie tak naprawdę podlegają te same dane:

```
>>> lista1[2]=3
>>> lista5
[1, 2, 3, 4]
>>> lista5[2]=5
>>> lista1
[1, 2, 5, 4]
```

Jeżeli listę utworzono z innych list:

```
>>> lista6=[0,lista1]
>>> lista6
[0, [1, 2, 5, 4]]
```

To każda zmiana ich wartości będzie przenoszona na listę nadrzędną:

```
>>> lista1[1]=4
>>> lista6
[0, [1, 4, 5, 4]]
```

Co więcej, zmiana wartości listy nadrzędnej będzie przenoszona na listę podrzędną:

```
>>> lista6[1][0]=2
>>> lista1
[2, 4, 5, 4]
```

5.13 Tworzenie i używanie krotek

Krotki pod wieloma względami przypominają listy, w podobny sposób tworzymy je i sprawdzamy ich wartości. W odróżnieniu od list, krotki są sekwencjami niezmiennymi, co powoduje różnice w sposobie ich modyfikacji.
Krotki tworzymy używając nawiasów okrągłych, rozdzielając ich elementy przecinkami.
Aby stworzyć krotkę z trzech liczb naturalnych, napiszemy:

```
>>> krotka1=(1,2,3)
>>> krotka1
(1, 2, 3)
```

Przy czym nawiasy okrągłe można pomijać:

```
>>> krotka1=1,2,3
>>> krotka1
(1, 2, 3)
```

(Dla zachowania przejrzystości my tego robić nie będziemy.)

Krotki mogą zawierać elementy różnych typów:

```
>>> krotka2=(1.0, 2, "trzy")
>>> krotka2
(1.0, 2, 'trzy')
```

W tym sekwencyjnych

```
>>> krotka3=(krotka1, lista1)
>>> krotka3
((1, 2, 3), [2, 4, 5, 4])
```

Krotki mogą być puste:

```
>>> krotka_pusta = ()
>>> krotka_pusta
()
>>> len (krotka_pusta)
0
```

Krotki mogą zawierać tylko jeden element. Jako że zapis (1) oznacza liczbę jeden w nawiasie, tworząc krotki jednoelementowe obowiązkowo na ich końcu stawiamy przecinek:

```
>>> liczba=(1)
```

```
>>> liczba
1
>>> krotka_jednoelementowa=(1,)
>>> krotka_jednoelementowa
(1,)
>>> len(krotka_jednoelementowa)
1
>>>
```

Można odczytywać wybiórczo zawartość poszczególnych elementów krotki:

```
>>> krotka1[1]
2
>>> krotka3[-1]
[2, 4, 5, 4]
```

Lub ich ciągów:

```
>>> krotka1[1:]
(2, 3)
>>> krotka1[::2] # co drugi element krotki
(1, 3)
```

Krotki można powielać:

```
>>> krotka1*=2
>>> krotka1
(1, 2, 3, 1, 2, 3)
```

Skracać:

```
>>> krotka1=krotka1[:3]
>>> krotka1
(1, 2, 3)
```

Wydłużać:

```
>>> krotka1=krotka1+(4,)
>>> krotka1
(1, 2, 3, 4)
```

5.14 Modyfikacja krotek

Krotki są sekwencjami niezmiennymi, więc nie można modyfikować ich fragmentów:

```
>>> krotka1[2]=1

Traceback (most recent call last):
  File "<pyshell#151>", line 1, in -toplevel-
    krotka1[2]=1
TypeError: object does not support item assignment
```

Oczywiście, pożądaną operację możemy wykonać, inaczej formułując polecenie:

```
>>> krotka1=krotka1[:2]+(1,)+krotka1[3:]
>>> krotka1
(1, 2, 1, 4)
```

5.15 Typ krotki jako typ niezmienny

Krotki są sekwencjami niezmiennymi, w związku z tym przypisanie krotki do krotki kopiuje faktyczne wartości, a nie jedynie wskaźnik do nich:

```
>>> krotka4=krotka1
>>> krotka4
```

```
(1, 2, 1, 4)
>>> krotka1=(1,2,3,4)
>>> krotka4
(1, 2, 1, 4)
```

5.16 Konwersja typów sekwencyjnych

W konwersji typów sekwencyjnych używamy następujących instrukcji:

- **list** zamienia typ sekwencyjny na listę

```
>>> lista7=list(krotka5)
>>> lista7
[2, 4, 5, 4]
>>> lista8=list("abcd")
>>> lista8
['a', 'b', 'c', 'd']
```

- **tuple** zamienia typ sekwencyjny na krotkę

```
>>> krotka5=tuple(lista1)
>>> krotka5
(2, 4, 5, 4)
>>> krotka6=tuple("abcd")
>>> krotka6
('a', 'b', 'c', 'd')
```

- **str** zamienia typ sekwencyjny na napis

```
>>> str(krotka6)
"('a', 'b', 'c', 'd')"
>>> str(lista7)
'[2, 4, 5, 4]'
```

Lub krócej, przy użyciu **odwróconych** apostrofów:

```
>>> `krotka6`
"('a', 'b', 'c', 'd')"
>>> `lista7`
'[2, 4, 5, 4]'
```

5.17 Pętle iterowane po elementach sekwencji

Aby wykonać jakieś operacje na wszystkich lub wybranych elementach sekwencji, najprościej jest posłużyć się pętlą **for ... in**.
Przykładowo, aby wyświetlić w kolejnych liniach wszystkie elementy sekwencji lista1 napiszemy:

```
>>> for a in lista1:
      print a


2
4
5
4
```

gdzie a jest przykładową zmienną, która w danym powtórzeniu pętli przyjmuje wartość kolejnych elementów sekwencji.

Aby poznać numer aktualnego powtórzenia pętli, posługujemy się funkcją **enumerate**:

```
>>> for nr, wartosc in enumerate(lista2):
      print nr,
      print wartosc
```

```
0 pierwszy
1 ostatni
```

gdzie nr jest zmienną zawierająca aktualny numer obiegu pętli, a wartosc zmienną, która w danym powtórzeniu pętli przyjmuje wartość kolejnych elementów sekwencji.

Możemy wykonywać pętlę tylko dla fragmentu sekwencji:

```
>>> for i in lista7[1:3]:
      print i,i**2


4 16
5 25
```

A wewnątrz pętli wykonywać instrukcje warunkowe:

```
>>> for i in lista7:
      if i>0:
            print i, i**0.5


2 1.41421356237
4 2.0
5 2.2360679775
4 2.0
```

5.18 Ćwiczenia kontrolne

I. Napisz program "parzyste2.py", który wczyta od użytkownika liczbę całkowitą i bez użycia instrukcji **if** wyświetli informację, czy jest to liczba parzysta, czy nieparzysta.

II. Napisz program "numer.py", który zamieni wprowadzony przez użytkownika ciąg cyfr na formę tekstową:
   a. znaki nie będące cyframi mają być ignorowane
   b. konwertujemy cyfry, nie liczby, a zatem:
      i. 911 to "dziewięć jeden jeden"
      ii. 1100 to "jeden jeden zero zero"

LEKCJA 6 – PĘTLE

6.1 Szybkie tworzenie sekwencji

Po uruchomieniu IDLE'a zgłasza się tryb interaktywny Pythona.
Wykorzystamy go do nauki szybkiego tworzenia sekwencji.
Do tworzenia sekwencji, których elementy należą do ciągu arytmetycznego, używamy funkcji **range**:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Pojedynczy parametr oznacza koniec (tj. pierwszy element nie należący do) sekwencji (pierwszym elementem zawsze jest zero).
Aby wyświetlić kwadraty liczb od 0 do 9, napiszemy:

```
>>> for x in range(10):
      print x,'** 2 =',x*x


0 ** 2 = 0
1 ** 2 = 1
2 ** 2 = 4
3 ** 2 = 9
4 ** 2 = 16
5 ** 2 = 25
6 ** 2 = 36
7 ** 2 = 49
8 ** 2 = 64
9 ** 2 = 81
```

Aby zmienić pierwszy element tworzonej sekwencji używamy funkcji **range** z dwoma parametrami (początek i koniec):

```
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Aby wyświetlić kwadraty liczb od 3 do 9, napiszemy:

```
>>> for x in range(3,10):
      print x,'** 2 =',x*x


3 ** 2 = 9
4 ** 2 = 16
5 ** 2 = 25
6 ** 2 = 36
7 ** 2 = 49
8 ** 2 = 64
9 ** 2 = 81
```

Aby zmienić krok pomiędzy elementami tworzonej sekwencji używamy funkcji **range** z trzema parametrami (początek, koniec i krok):

```
>>> range(1,10,2)
[1, 3, 5, 7, 9]
```

Możemy w ten sposób również odwrócić kolejność elementów, poprzez użycie ujemnego kroku:

```
>>> range(9,0,-1)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Aby wyświetlić kwadraty liczb nieparzystych malejąco od 9 do 1, napiszemy:

```
>>> for x in range(9,0,-2):
      print x,'** 2 =',x*x


9 ** 2 = 81
7 ** 2 = 49
5 ** 2 = 25
3 ** 2 = 9
1 ** 2 = 1
```

6.2 Formatowanie liczb

Celem pętli często jest wyświetlenie kolumny liczb. Aby liczby wyświetlane były w należyty sposób i w pożądanym miejscu używamy operatora formatowania **%** w połączeniu z ciągiem formatującym. Ciąg formatujący składa się ze znaku **%**, po którym następują opcje formatowania, ilość znaków przeznaczonych do wyświetlenia oraz typ danej do wyświetlenia (przy czym tylko trzeci element – tj. typ danych jest wymagany).
Typ danej sygnalizujemy pojedynczą literą. I tak:

- Litera **s** oznacza napis (konwertuje każdy typ danych na tekst), np.:

```
>>> print "%s" % 1
1
>>> print "%s" % range(6)
[0, 1, 2, 3, 4, 5]
>>> print "%s" % "txt"
txt
```

- Litera **c** oznacza pojedynczy znak w kodzie ASCII, np.:

```
>>> print "%c" % "A"
A
>>> print "%c" % 077
?
>>> print "%c" % 33
!
>>> print "%c" % "Abc" # błąd!

Traceback (most recent call last):
  File "<pyshell#44>", line 1, in -toplevel-
    print "%c" % "Abc"
TypeError: %c requires int or char
```

- Litera **i** oznacza dziesiętną liczbę całkowitą (konwertuje kompatybilny typ danych na liczbę całkowitą), np.:

```
>>> print "%i" % 0xff
255
>>> print "%i" % 2.2
2
>>> print "%i" % "11" # błąd!

Traceback (most recent call last):
  File "<pyshell#32>", line 1, in -toplevel-
    print "%i" % "11" # błąd!
TypeError: int argument required
```

- Litera **x** oznacza szesnastkową liczbę całkowitą (konwertuje kompatybilny typ danych na liczbę całkowitą), np.:

```
>>> print "%x" % 0xff
ff
>>> print "%x" % 2.2
2
>>> print "%x" % 22
16
```

- Litera **o** oznacza ósemkową liczbę całkowitą (konwertuje kompatybilny typ danych na liczbę całkowitą), np.:

```
>>> print "%o" % 0xff
377
>>> print "%o" % 2.2
2
>>> print "%o" % 077
77
```

- Litera **e** oznacza liczbę zmiennopozycyjną w postaci wykładniczej, np.:

```
>>> print "%e" % 1
1.000000e+000
>>> print "%e" % 1.23
1.230000e+000
>>> print "%e" % 123
1.230000e+002
```

- Litera **f** oznacza liczbę zmiennopozycyjną w postaci ułamka dziesiętnego, np.:

```
>>> print "%f" % 123
123.000000
>>> print "%f" % 1.23
1.230000
```

6.3 Ustalenie długości pola do wyświetlenia tekstu

Aby przekonać się na czym polega zaleta formatowania, wyświetlmy tabelę kwadratów i sześcianów wybranych liczb:

```
>>> for x in range(5,100,10):
      print x,x**2,x**3


5 25 125
15 225 3375
25 625 15625
35 1225 42875
45 2025 91125
55 3025 166375
65 4225 274625
75 5625 421875
85 7225 614125
95 9025 857375
```

Jak widać kolumny liczb wyświetlane są nierówno. Spróbujmy ustalić w formacie długość pola do wyświetlenia każdej liczby na 4, jej kwadratu na 6, a sześcianu na 8:

```
>>> for x in range(5,100,10):
      print "%4i%6i%8i" % (x,x**2,x**3)


   5    25     125
  15   225    3375
  25   625   15625
  35  1225   42875
  45  2025   91125
  55  3025  166375
  65  4225  274625
  75  5625  421875
  85  7225  614125
  95  9025  857375
```

Jak widać, efekt teraz jest znacznie bardziej przejrzysty.
Formatując liczby zmiennopozycyjne możemy także ustalić nie tylko całkowitą długość, ale także liczbę wyświetlanych miejsc po przecinku (np. na 3):

```
>>> for x in range(5,100,10):
      print "Pierwiastkiem liczby %2i jest %5.3f" % (x,x**0.5)


Pierwiastkiem liczby  5 jest 2.236
Pierwiastkiem liczby 15 jest 3.873
Pierwiastkiem liczby 25 jest 5.000
Pierwiastkiem liczby 35 jest 5.916
Pierwiastkiem liczby 45 jest 6.708
Pierwiastkiem liczby 55 jest 7.416
Pierwiastkiem liczby 65 jest 8.062
Pierwiastkiem liczby 75 jest 8.660
Pierwiastkiem liczby 85 jest 9.220
Pierwiastkiem liczby 95 jest 9.747
```

6.4 Opcje formatowania

Opcje formatowania modyfikują sposób wyświetlania liczb. Np. opcja **+** wymusza wyświetlanie znaku liczby, także dla liczb nieujemnych:

```
>>> for x in range (-10,11):
      print "%+i" % x,


-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +10
```

Załóżmy, że chcemy otrzymać tabelę przeliczającą liczby dziesiętne na ósemkowe i szesnastkowe:

```
>>> for x in range(5,100,10):
      print "%3i%6o%5x" % (x,x,x)


  5     5    5
 15    17    f
 25    31   19
 35    43   23
 45    55   2d
 55    67   37
 65   101   41
 75   113   4b
 85   125   55
 95   137   5f
```

Nie jest to jednoznaczne. Użycie opcji **#** spowoduje, że liczby ósemkowe i szesnastkowe będą poprzedzane właściwym prefiksem:

```
>>> for x in range(5,100,10):
      print "%3i%#6o%#5x" % (x,x,x)


  5    05  0x5
 15   017  0xf
 25   031 0x19
 35   043 0x23
 45   055 0x2d
 55   067 0x37
 65  0101 0x41
 75  0113 0x4b
 85  0125 0x55
 95  0137 0x5f
```

Z kolei użycie opcji **-** spowoduje, że liczby będą wyrównywane do lewej, a nie prawej krawędzi swojego pola:

```
>>> for x in range(5,100,10):
```

```
      print "%-3i%#-6o%#-5x" % (x,x,x)


5  05    0x5
15 017   0xf
25 031   0x19
35 043   0x23
45 055   0x2d
55 067   0x37
65 0101  0x41
75 0113  0x4b
85 0125  0x55
95 0137  0x5f
```

Natomiast użycie opcji **0** spowoduje, że pole przeznaczone na liczby będzie wypełniane nie spacjami, lecz zerami:

```
>>> for x in range(5,100,10):
      print "%3i %#04o %#04x" % (x,x,x)


  5 0005 0x05
 15 0017 0x0f
 25 0031 0x19
 35 0043 0x23
 45 0055 0x2d
 55 0067 0x37
 65 0101 0x41
 75 0113 0x4b
 85 0125 0x55
 95 0137 0x5f
```

6.5 Pętle zagnieżdżone

Pętle mogą zawierać inne pętle – mówimy wtedy o nich, że są zagnieżdżone.
Spróbujmy wygenerować tabliczkę mnożenia:

```
>>> 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
```

6.6 Zmiana przebiegu pętli

Nie wszystkie instrukcje w pętli muszą być wykonane za każdym razem. Do pomijania wszystkich instrukcji znajdujących się po niej w pętli służy instrukcja **continue**.
Załóżmy, że chcemy napisać program, który wyliczy i wyświetli pierwiastek kwadratowy dla każdej nieujemnej liczby z listy (lub krotki) podanej przez użytkownika.
Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.**
Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **pierwiastki.py**.

Przepiszmy następujący program:

```
# Program, który wyliczy i wyświetli pierwiastek kwadratowy
# dla każdej nieujemnej liczby z listy podanej przez użytkownika
liczby = input("Podaj kilka liczb:")
for x in liczby:
   if x<0: continue
   print "Pierwiastkiem liczby %2i jest %5.3f" % (x,x**0.5)
```

W celu uruchomienia programu wybieramy z menu **Run** polecenie **Run Module** (lub wciskamy klawisz **F5**). Po chwili nasz program zostanie uruchomiony. Wypróbujmy:

```
>>>
Podaj kilka liczb:1,-1,2
Pierwiastkiem liczby  1 jest 1.000
Pierwiastkiem liczby  2 jest 1.414
```

Nie wszystkie zaplanowane obiegi pętli muszą być wykonane za każdym razem. Do przerywania pętli w dowolnym miejscu służy instrukcja **break**.
Załóżmy, że chcemy napisać program, który znajdzie i wyświetli pozycję na liście (lub w krotce) podanej przez użytkownika pierwszego wystąpienia liczby również podanej przez użytkownika.
Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.**
Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **szukaj.py**.

```
# Program, który znajdzie i wyświetli pozycję na liście
# pierwszego wystąpienia określonej liczby
liczby = input("Podaj kilka liczb:")
szukana = input("Podaj liczbę do znalezienia:")
for p,x in enumerate(liczby):
   if x != szukana: continue
   print "Znaleziono liczbę %i na pozycji %i" % (x,p+1)
```

W celu uruchomienia programu wybieramy z menu **Run** polecenie **Run Module** (lub wciskamy klawisz **F5**). Po chwili nasz program zostanie uruchomiony. Wypróbujmy:

```
Podaj kilka liczb:1,2,3,2
Podaj liczbę do znalezienia:2
Znaleziono liczbę 2 na pozycji 2
Znaleziono liczbę 2 na pozycji 4
>>>
```

Jak widać, w tej chwili program wypisuje wszystkie, a nie tylko pierwszą pozycję szukanej liczby. Dopiszmy zatem na końcu programu jedną linię (pamiętajmy o wcięciu!):

```
   break
```

W celu uruchomienia programu wybieramy z menu **Run** polecenie **Run Module** (lub wciskamy klawisz **F5**). Po chwili nasz program zostanie uruchomiony. Wypróbujmy:

```
Podaj kilka liczb:1,2,3,2
Podaj liczbę do znalezienia:2
Znaleziono liczbę 2 na pozycji 2
>>>
```

Jak widać, w tej chwili program wypisuje już tylko pierwszą pozycję szukanej liczby.
Jeżeli jednak podamy liczbę, której nie ma na liście, nic nie zobaczymy:

```
Podaj kilka liczb:1,2
Podaj liczbę do znalezienia:3
>>>
```

Aby to naprawić, posłużymy się instrukcją **else**. Instrukcja ta określa co ma się wykonać na zakończenie pętli, ale tylko wtedy, gdy nie przerwano pętli instrukcją **break**.
Dopiszmy jeszcze dwie linie na końcu programu **szukaj.py:**

```
else:
    print "Liczby %i nie ma na liście" % szukana
```

Wypróbujmy:

```
Podaj kilka liczb:1,2
Podaj liczbę do znalezienia:5
Liczby 5 nie ma na liście
>>>

Podaj kilka liczb:1,3
Podaj liczbę do znalezienia:3
Znaleziono liczbę 3 na pozycji 2
>>>
```

6.7 Pętle o nieznanej liczbie powtórzeń

Wiemy już, jak przerwać we właściwym miejscu zbyt długą pętlę. Czasami jednak musimy użyć w programie pętli, której liczby powtórzeń nie jesteśmy w stanie w żaden sposób przewidzieć. Przypomnijmy sobie algorytm Euklidesa na wyliczanie Największego Wspólnego Dzielnika. Powtarzamy w nim operacje dzielenia, nie jesteśmy jednak w stanie z góry powiedzieć, ile tych operacji będzie.
Do tworzenia pętli o nieznanej liczbie powtórzeń w Pythonie służy instrukcja **while**.
Przejdźmy do okna trybu interaktywnego i wpiszmy:

```
>>> a=[1,2,3,4,5,6]
>>> while a:
      a=a[:len(a)-1]
      print a


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

Pętla typu **while** może również zawierać blok po **else**, wykonywany po ostatnim obiegu pętli:

```
>>> a=7
>>> while a:
      a-=1
      print a
else: # wciśnij klawisz backspace by cofnąć wcięcie
      print "koniec"


6
5
4
3
2
1
0
koniec
```

6.8 Przykład: wyliczanie Największego Wspólnego Dzielnika

Spróbujemy teraz napisać program na wyliczanie metodą Euklidesa NWD dwóch liczb podanych przez użytkownika.
Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno,

przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **nwd.py**.

```
# Wyliczanie NWD i NWW
# 1. wprowadzanie liczb
print "Podaj dwie liczby naturalne:”
a = input ("Pierwsza:")
b = input ("Druga:")
# 2. ustalenie która 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. wyświetlenie rezultatów
print "NWD liczb %i i %i wynosi %i, a ich NWW wynosi %i" % (a,b,m,a*b/m)
```

Wypróbujmy:

```
>>> ================================ RESTART ================================
>>>
Podaj dwie liczby naturalne:
Pierwsza:6
Druga:8
NWD liczb 6 i 8 wynosi 2, a ich NWW wynosi 24
>>> ================================ RESTART ================================
>>>
Podaj dwie liczby naturalne:
Pierwsza:21
Druga:14
NWD liczb 21 i 14 wynosi 7, a ich NWW wynosi 42
>>>
```

6.9 Przykład: wyszukiwanie liczb pierwszych

Do wyszukania liczb pierwszych z podanego zakresu posłużymy się sitem Eratostenesa.
Algorytm ten polega na usuwaniu z badanego zakresu wszystkich wielokrotności kolejnych liczb pierwszych.
Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **pierwsze.py**.

```
# Wyszukiwanie liczb pierwszych
koniec = input("Podaj górną granicę zakresu do wyszukania liczb pierwszych:")
pierwsze = range(koniec+1)
n = 1
while ( n < koniec ):
        n += 1
        if  pierwsze[n]==0:  # zerem oznaczamy liczby nie-pierwsze
                                                continue
        m = n * 2
        while m <= koniec:
           pierwsze[m]=0  # zerem oznaczamy liczby nie-pierwsze
           m += n
print "Znaleziono następujące liczby pierwsze:"
for  n  in  pierwsze[2:]:
     if  n:  print  n,
```

Wypróbujmy:

```
Podaj górną granicę zakresu do wyszukania liczb pierwszych:99
Znaleziono następujące liczby pierwsze:
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
>>>
```

6.10 Ćwiczenia kontrolne

I. Woda zamarza przy 32 stopniach Fahrenheita, a wrze przy 212 stopniach Fahrenheita. Napisz program „stopnie.py", który wyświetli tabelę przeliczeń stopni Celsjusza na stopnie Fahrenheita w zakresie od –20 do +40 stopni Celsjusza (co 5 stopni). Pamiętaj o wyświetlaniu znaku plus/minus przy temperaturze.

II. Napisz program „oceny.py", który wczytuje od użytkownika kolejne oceny i:
- sprawdza czy wprowadzona ocena jest na liście dopuszczalnych na wydziale ocen
  - jeżeli ocena jest na liście dopuszczalnych na wydziale ocen, dodaje ją na listę otrzymanych ocen
- jeżeli wciśnięto sam Enter, oznacza to koniec listy otrzymanych ocen
- wyświetla wyliczoną dla listy otrzymanych ocen średnią arytmetyczną.

LEKCJA 7 - OBIEKTY, METODY, MODUŁY, FUNKCJE MATEMATYCZNE

7.1 Podstawy podejścia obiektowego

Włączmy tryb interaktywny Pythona.
Python jest językiem zorientowanym obiektowo. By w pełni korzystać z jego możliwości, musimy zatem nauczyć się podstaw podejścia obiektowego.
Zasadniczą koncepcją w podejściu obiektowym do programowania jest połączenie w całość danych oraz algorytmów, które na tych danych operują. Takie połączenie danych i algorytmów nazywamy obiektem.
Obiekt posiada pewne własności, czyli dane oraz pewne metody, czyli algorytmy do przetwarzania tych danych.
W języku Python dostęp do właściwości i metod określonego obiektu uzyskujemy stawiając kropkę po jego nazwie.
Metody od właściwości odróżniamy po tym, że jako funkcje muszą mieć na końcu nawiasy okrągłe.
Zbiór obiektów o tych samych własnościach i metodach nazywamy klasą.

Dla przykładu stwórzmy zmienną zespoloną z:

```
>>> z=3+2j
>>> z
(3+2j)
```

Klasa liczb zespolonych **complex** posiada dwie właściwości **real** i **imag** przechowujące część rzeczywistą i część urojoną liczby:

```
>>> z.real
3.0
>>> z.imag
2.0
```

Klasa liczb zespolonych **complex** posiada także metodę **conjugate( )** pozwalającą wyliczyć liczbę sprzężoną do przechowywanej:

```
>>> z.conjugate()
(3-2j)
>>> z*z.conjugate()
(13+0j)
```

7.2 Metody operujące na napisach

Wyjątkowo dużo metod posiada klasa **string**. Metody te służą do różnego typu konwersji i formatowania napisów.
Na początek stwórzmy zmienną napisową s:

```
>>> s="to jest NAPIS"
>>> s
'to jest NAPIS'
```

Metoda **capitalize( )** służy do nadania napisowi formatu jak w zdaniu, to jest zmiany pierwszej litery na dużą, a pozostałych na małe:

```
>>> s.capitalize()
'To jest napis'
```

Metoda **center( )** służy do wyśrodkowania napisu w polu o podanej długości. Domyślnie pole dopełniane jest znakiem spacji:

```
>>> s.center(32)
'         to jest NAPIS          '
>>> s.center(64)
'                         to jest NAPIS                          '
```

Możemy jednak podać inny znak wypełnienia (jako drugi parametr metody):

```
>>> s.center(64,'*')
'************************to jest NAPIS*************************'
```

Metoda **count( )** oblicza ile razy określony ciąg znaków występuje w napisie:

```
>>> s.count('t')
2
>>> (s*10).count(s)
10
```

Metoda **find( )** odnajduje pierwsze wystąpienie określonego ciąg znaków w napisie:

```
>>> s.find('NAPIS')
8
```

Jeżeli szukanego ciągu w napisie nie ma, zwracana jest wartość -1:

```
>>> s.find('napis')
-1
```

Metoda **isdigit( )** sprawdza, czy napis zawiera tylko cyfry:

```
>>> s.isdigit( )
False
>>> '18'.isdigit()
True
>>> '18.2'.isdigit()
False
```

Metoda **join( )** łączy wszystkie elementy sekwencji podanej jako parametr w pojedynczy napis, wstawiając pomiędzy nie napis dla którego wywołujemy metodę:

```
>>> ' '.join(['ala','ma','kota'])
'ala ma kota'
>>> ','.join(['ala','ma','kota'])
'ala,ma,kota'
>>> s.join(['***']*5)
'***to jest NAPIS***to jest NAPIS***to jest NAPIS***to jest NAPIS***'
```

Metoda **lower( )** zamienia wszystkie duże litery w napisie na małe:

```
>>> s.lower()
'to jest napis'
```

Metoda **replace( )** zamienia wszystkie wystąpienia określonego ciąg znaków w napisie na inny ciąg:

```
>>> s.replace('NAPIS','tekst')
'to jest tekst'
>>> s.replace(' ','---')
'to---jest---NAPIS'
```

Metoda **rfind( )** odnajduje ostatnie wystąpienie określonego ciąg znaków w napisie:

```
>>> s.rfind('NAPIS')
8
>>> 'ala ma kota'.rfind('a')
10
```

Jeżeli szukanego ciągu w napisie nie ma, zwracana jest wartość -1:

```
>>> s.rfind('napis')
-1
```

Metoda **rjust( )** służy do wyrównania napisu do prawej w polu o podanej długości. Domyślnie pole dopełniane jest znakiem spacji:

```
>>> s.rjust(32)
'                   to jest NAPIS'
>>> s.rjust(64)
'                                                to jest NAPIS'
```

Możemy jednak podać inny znak wypełnienia (jako drugi parametr metody):

```
>>> s.rjust(64,'.')
'.............................................to jest NAPIS'
```

Metoda **split()** tworzy listę wyrazów występujących w napisie:

```
>>> s.split()
['to', 'jest', 'NAPIS']
>>> for wyraz in s.split():
      print wyraz.capitalize().rjust(60)


                                                          To
                                                        Jest
                                                       Napis
```

Jako parametr możemy podać znak rozdzielający wyrazy (domyślnie spacja):

```
>>> '032-345-231'.split('-')
['032', '345', '231']
```

Metoda **splitlines()** tworzy listę linii występujących w napisie:

```
>>> ((s+'\n')*10).splitlines()
['to jest NAPIS', 'to jest NAPIS', 'to jest NAPIS', 'to jest NAPIS', 'to jest NAPIS',
'to jest NAPIS', 'to jest NAPIS', 'to jest NAPIS', 'to jest NAPIS', 'to jest NAPIS']
```

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

```
>>> s.swapcase()
'TO JEST napis'
```

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

```
>>> s.title( )
'To Jest Napis'
```

Metoda **upper( )** zamienia wszystkie małe litery w napisie na duże:

```
>>> s.upper()
'TO JEST NAPIS'
```

7.3 Metody operujące na listach

Również klasa **list** posiada dużo metod. Metody te służą do różnego typu modyfikacji i porządkowania list.
Na początek stwórzmy listę l:

```
>>> l=range(1,21)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Metoda **append( )** dołącza do listy pojedynczy element:

```
>>> l.append(33)
```

```
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 33]
```

Metoda **extend( )** dołącza do listy inną listę:

```
>>> l.extend([33,99])
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 33, 33, 99]
>>>
```

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

```
>>> l.count(33)
2
>>> l.count(99)
1
>>> l.count(102)
0
```

Metoda **index(*w*)** znajduje pierwszą pozycję listy na której występuje wartość ***w***:

```
>>> l.index(33)
20
```

Można ograniczyć przeszukiwanie do części listy przez podanie dodatkowo dwóch parametrów - początku i końca zakresu:

```
>>> l.index(33,21,22)
21
```

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

```
>>> l.insert(5,77)
>>> l
[1, 2, 3, 4, 5, 77, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 33, 33, 99]
```

Metoda **pop(*i*)** zwraca wartość z pozycji *i* listy, po czym usuwa tę pozycję:

```
>>> l.pop(5)
77
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 33, 33, 99]
>>>
```

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

```
>>> l.remove(99)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 33, 33]
>>> l.count(33)
2
>>> l.remove(33)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 33]
>>> l.count(33)
1
>>> l.remove(33)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> l.count(33)
0
```

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

```
>>> l.reverse()
```

```
>>> l
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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

```
>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

7.4 Moduły

Moduły Pythona zawierają definicje rzadziej używanych funkcji i typów danych.
Dostęp do nich uzyskujemy dzięki instrukcji **import**. Wpiszmy:

```
>>> import random
```

Od tej pory mamy dostęp do zawartości modułu **random**. Moduł ten zawiera funkcje obsługujące generowanie liczb pseudolosowych:

```
>>> random.seed()
```

Inicjalizuje generator liczb pseudolosowych. Użycie tej funkcji powinno zawsze poprzedzać losowanie jakiejkolwiek liczby.

```
>>> random.randint(1,10)
3
>>> random.randint(1,10)
1
>>> random.randint(1,10)
4
```

Funkcja **randint**(*od*, *do*) losuje liczbę całkowitą z zakresu *od*..*do*.

Zauważmy, że nazwę funkcji poprzedzamy nazwą modułu i kropką.
Otrzymamy błąd, jeżeli napiszemy po prostu:

```
>>> randint(1,10)

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in -toplevel-
    randint(1,10)
NameError: name 'randint' is not defined
```

Możemy jednak uczynić nazwy funkcji dostępnymi bez potrzeby używania nazwy modułu poprzez użycie instrukcji **from**:

```
>>> from random import randint
```

Od tej pory możemy po prostu napisać:

```
>>> randint(1,10)
6
```

Ale jeżeli napiszemy

```
>>> choice(s.split())

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in -toplevel-
    choice(s.split())
NameError: name 'choice' is not defined
```

Zamiast udostępniać po kolei nazwy poszczególnych funkcji, można zażądać wszystkich nazw (ale uwaga! nie dla każdego modułu działa to prawidłowo):

```
>>> from random import *
```

I dalej:

```
>>> choice(s.split())
'to'
```

Funkcja **choice** wybiera losowy element z sekwencji.

```
>>> choice(s.split())
'jest'
>>> choice(s)
'o'
>>> choice(l)
1
```

Funkcja **shuffle** wykonuje losową permutację sekwencji:

```
>>> shuffle(l)
>>> l
[11, 20, 1, 9, 10, 2, 14, 18, 5, 6, 15, 3, 7, 16, 17, 12, 8, 4, 19, 13]
>>> shuffle(l)
>>> l
[15, 1, 13, 14, 9, 8, 4, 10, 18, 16, 12, 3, 11, 17, 19, 7, 5, 6, 20, 2]
>>>
```

Funkcja **random** zwraca losową liczbę rzeczywistą z przedziału [0.0, 1.0):

```
>>> random()
0.1964729982262956
```

Funkcja **uniform(*a,b*)** zwraca losową liczbę rzeczywistą z przedziału [*a*, *b*):

```
>>> uniform(10,20)
13.952472157445552
```

Funkcja **uniform(*a,b*)** zwraca losową liczbę rzeczywistą z przedziału [*a*, *b*):

```
>>> uniform(10,20)
13.952472157445552
```

Oprócz jednostajnego, dostępne są i inne rozkłady zmiennych losowych. Np. funkcja **normalvariate**(*mu*, *sigma*) zwraca wartość zmiennej losowej o rozkładzie normalnym, o średniej *mu* i odchyleniu standardowym *sigma*:

```
>>> normalvariate(10,5)
4.6557086378024239
```

7.5 Funkcje matematyczne

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

```
>>> from math import *
```

Funkcje modułu **math** operują na liczbach rzeczywistych.
Funkcja **math.ceil**(*x*) zwraca sufit liczby rzeczywistej *x* (najmniejszą liczbę całkowitą nie mniejszą niż *x*)

```
>>> ceil(2.7)
3.0
```

Funkcja **math.fabs**(*x*) zwraca wartość absolutną liczby rzeczywistej *x*

```
>>> fabs(-3)
3.0
```

Funkcja **math.floor**(*x*) zwraca podłogę liczby rzeczywistej *x* (największą liczbę całkowitą nie większą niż *x*)

```
>>> floor(2.7)
2.0
```

Funkcja **math.modf**(*x*) zwraca krotkę zawierającą część ułamkową i całkowitą liczby rzeczywistej *x*

```
>>> modf(2.5)
(0.5, 2.0)
>>> modf(2.5)[0]
0.5
```

Funkcja **math.exp**(*x*) zwraca *e* do potęgi *x*

```
>>> exp(1)
2.7182818284590451
```

Funkcja **math.log**(*x*) zwraca logarytm naturalny z *x*

```
>>> log(e)
1.0
```

By zmienić podstawę logarytmu podajemy drugi parametr funkcji **math.log**

```
>>> log(256,2)
8.0
```

Funkcja **math.sqrt**(*x*) zwraca pierwiastek kwadratowy z *x*

```
>>> sqrt(49)
7.0
```

Funkcja **math.acos**(*x*) zwraca w radianach arcus cosinus kąta *x*

```
>>> acos(1)
0.0
```

Funkcja **math.asin**(*x*) zwraca w radianach arcus sinus kąta *x*

```
>>> asin(0)
0.0
```

Funkcja **math.atan**(*x*) zwraca w radianach arcus tangens kąta *x*

```
>>> atan(0)
0.0
```

Funkcja **math.cos**(*x*) zwraca cosinus kąta *x* podanego w radianach

```
>>> cos(1)
0.0
```

Funkcja **math.sin**(*x*) zwraca sinus kąta *x* podanego w radianach

```
>>> sin(0)
0.0
```

Funkcja **math.tan**(*x*) zwraca tangens kąta *x* podanego w radianach

```
>>> tan(0)
0.0
```

Funkcja **math.hypot**(*x, y*) zwraca odległość punktu o współrzędnych (*x, y*) od początku układu (0, 0) (długość przeciwprostokątnej dla przyprostokątnych o długościach *x* i *y*)

```
>>> hypot(3,4)
5.0
```

Funkcja **math.degrees**(*x*) zamienia miarę kąta x wyrażoną w radianach na stopnie

```
>>> degrees(pi)
180.0
```

Funkcja **math.radians**(*x*) zamienia miarę kąta x wyrażoną w stopniach na radiany

```
>>> radians(180)
3.1415926535897931
```

Jak można było zauważyć w przykładach, moduł **math** definuje także dwie stałe: pi oraz e.

7.6 Przykład: pisanie wyrazów wspak

Spróbujemy teraz napisać program na odwracanie poszczególnych wyrazów zdania. Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **wspak.py**.

```
# program wyświetla poszczególne wyrazy napisu wspak
t=raw_input("Wpisz dłuższy tekst >")
for w in t.split():      # dla każdego wyrazu w zdaniu
    l=list(w)            # tworzymy listę jego liter
    l.reverse()          # odwracamy jej kolejność
    print ''.join(l),    # łączymy ją w całość i wyświetlamy bez separatorów
```

Wypróbujmy:

```
>>> ================================ RESTART ================================
>>>
Wpisz dłuższy tekst >dzieckiem w kolebce kto łeb urwał hydrze
meikceizd w ecbelok otk beł ławru ezrdyh
>>>
```

7.7 Przykład: wyliczanie odległości między dwoma punktami na płaszczyźnie

Spróbujemy teraz napisać program do wyliczania odległości między dwoma punktami na płaszczyźnie. Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **punkty.py**.

```
# program wylicza odległość między dwoma punktami
from math import hypot
p1x,p1y = input ("Podaj współrzędne poziomą i pionową pierwszego punktu >")
p2x,p2y = input ("Podaj współrzędne poziomą i pionową drugiego punktu >")
print "Odległość między tymi punktami wynosi %.3f" % hypot(p1x-p2x,p1y-p2y)
```

Wypróbujmy:

```
>>> ================================ RESTART ================================
>>>
Podaj współrzędne poziomą i pionową pierwszego punktu >0,1
Podaj współrzędne poziomą i pionową drugiego punktu >1,0
Odległość między tymi punktami wynosi 1.414
>>> ================================ RESTART ================================
>>>
Podaj współrzędne poziomą i pionową pierwszego punktu >-1,-2
Podaj współrzędne poziomą i pionową drugiego punktu >3,1
```

```
Odległość między tymi punktami wynosi 5.000
>>>
```

7.8 Ćwiczenia kontrolne

I. Napisz program "tryg.py", który wczyta od użytkownika wielkość kąta w stopniach i wyświetli wartość czterech podstawowych funkcji trygonometrycznych (sin, cos, tg, ctg) o ile dla danego kąta jest to możliwe.

II. Napisz program "lotto.py", który wyświetli 6 losowych i nie powtarzających się liczb z zakresu od 1 do 49.

III. Napisz program "wyrazy.py", który wczyta od użytkownika pewien tekst, a następnie podzieli go na zdania (zakładamy, że jednoznacznie kropka rozdziela zdania) i dla każdego zdania wyświetli ile jest w nim wyrazów (zakładamy, że spacja oddziela wyrazy w zdaniu).

LEKCJA 8 - DEFINIOWANIE FUNKCJI W PYTHONIE

8.1 Definiowanie funkcji

Poznaliśmy dotąd szereg funkcji Pythona, zarówno wbudowanych, jak i dostępnych z poziomu dołączanych modułów.
Jak w każdym języku umożliwiającym programowanie strukturalne, Python pozwala również tworzyć programiście własne funkcje.
Definicja funkcji musi zawierać:

- nagłówek funkcji obejmujący
  - nazwę funkcji, która pozwoli zidentyfikować funkcję w pozostałej części programu
  - listę argumentów, która funkcja otrzymuje na początku działania programu
- ciało funkcji, zawierające instrukcje, które zostaną wykonane w momencie wywołania (użycia) funkcji
  - jeżeli funkcja ma zwracać jakiś rezultat, musi zawierać odpowiednią instrukcję

W języku Python składnia definicji funkcji jest następująca:
def nazwa_funkcji (lista_parametrów):
    instrukcje_do_wykonania

Przejdźmy do trybu interaktywnego Pythona.
Zdefiniujemy przykładową funkcję pierw, która wyliczać będzie pierwiastek kwadratowy liczby rzeczywistej podanej jako argument.

```
>>> def pierw(n):
      return n**0.5
```

Jak widać funkcja jest bardzo prosta, a jej ciało składa się tylko z jednej linii zawierającej instrukcję **return**.
Instrukcja **return** służy do przekazywania rezultatu na zewnątrz funkcji. Wyrażenie po niej zostanie wyliczone, a jego wartość zwrócona jako rezultat funkcji.
Wypróbujmy działanie funkcji:

```
>>> pierw(2)
1.4142135623730951
>>> pierw(9)
3.0
```

8.2 Usuwanie i redefiniowanie funkcji

Zdefiniowaną uprzednio funkcję możemy w dowolnym miejscu usunąć, posługując się instrukcją **del**. A zatem:

```
>>> del pierw
```

I teraz:

```
>>> pierw(3)

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    pierw(3)
NameError: name 'pierw' is not defined
>>>
```

Co oznacza, że funkcję pierw udało nam się usunąć i nie możemy dalej jej używać.
Nie martwmy się jednak tą stratą. Zaraz zdefiniujemy nową, lepszą funkcję pierw.
Uwzględni ona, że pierwiastek kwadratowy możemy obliczać tylko dla nieujemnych liczb rzeczywistych.

```
>>> def pierw(n):
      if n>=0: return n**0.5
```

Dla liczb nieujemnych funkcja działa prawidłowo:

```
>>> pierw(3)
1.7320508075688772
```

A dla ujemnych nie powoduje komunikatu o błędzie

```
>>> pierw(-3)
>>>
```

Aby zmienić funkcję, nie musimy jej wpierw kasować. Wystarczy, że od nowa ją zdefiniujemy.
Poprawmy naszą funkcję tak, aby wyliczała pierwiastek również dla ujemnych liczb rzeczywistych:

```
>>> def pierw(n):
      if n>=0: return n**0.5
      else: return (-n)**0.5*1j
```

Sprawdźmy:

```
>>> pierw(4)
2.0
>>> pierw(-4)
2j
```

8.3 Parametry formalne i aktualne, zmienne lokalne

Występujący w nagłówku funkcji identyfikator n nazywamy parametrem formalnym. Jest to nazwa, pod którą przekazana do funkcji wartość widziana jest wewnątrz ciała funkcji.
Parametr formalny jest szczególnym rodzajem zmiennej lokalnej (szczególnym, bo inicjalizowanym wartością podaną przy wywołaniu w nawiasach). Zmienna lokalna to każda zmienna inicjowana w obrębie funkcji. Zmienna lokalna:

- dostępna jest tylko w obrębie funkcji; a zatem:

```
>>> pierw(7)
2.6457513110645907
>>> n

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    n
NameError: name 'n' is not defined
```

- "przykrywa" zmienną o tej samej nazwie istniejącą poza funkcją

```
>>> n=8
>>> pierw(7)
2.6457513110645907
>>> n
8
```

Jak widać modyfikacja zmiennej lokalnej n (na 7), nie zmienia wartości zmiennej globalnej n (nadal 8).

Występująca w wywołaniu funkcji wartość 7 to parametr aktualny. Parametry aktualne to faktyczne wartości przekazane do funkcji.
W momencie wywołania funkcji wszystkie operacje przewidziane do wykonania na parametrze formalnym, wykonywane są na parametrze aktualnym (czyli w tym przypadku na liczbie 7).

8.4 Wiele argumentów, wiele rezultatów

Funkcja może przyjmować więcej niż jeden argument i zwracać więcej niż jeden rezultat.

Poniżej mamy przykładową funkcję rs, która dla dwóch liczb zwraca ich sumę oraz różnicę.

```
>>> def rs(a,b):
      return a+b,a-b
```

Wypróbujmy:

```
>>> rs(1,3)
(4, -2)
```

Jak widać, rezultat wywołania funkcji, która zwraca więcej niż jedną wartość, jest krotką.
Możemy to wykorzystać w iteracji:

```
>>> for n in rs(3,4): print n

7
-1
```

Lub skonwertować wynik na listę:

```
>>> list(rs(2,7))
[9, -5]
```

Jeżeli wartości, które mają zostać przekazane jako argumenty funkcji zawarte są w sekwencji, np.:

```
>>> l=[2,3]
```

nie da się bezpośrednio przekazać takiej sekwencji jako listy argumentów (gdyż traktowana jest ona jako pojedynczy argument):

```
>>> rs(l)

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in -toplevel-
    rs(l)
TypeError: rs() takes exactly 2 arguments (1 given)
```

o ile nie "rozpakujemy" elementów sekwencji przy użyciu gwiazdki:

```
>>> rs(*l)
(5, -1)
```

8.5 Domyślne i nazwane wartości argumentów

Zdefiniujmy następującą funkcję:

```
>>> def kto(imie,wiek,jaki):
      print imie,"mimo swych",wiek,"lat jest bardzo",jaki+"m człowiekiem"
```

Wypróbujmy ją:

```
>>> kto("Adam",12,"mądry")
Adam mimo swych 12 lat jest bardzo mądrym człowiekiem
>>> kto("Ola",16,"rozsądny")
Ola mimo swych 16 lat jest bardzo rozsądnym człowiekiem
```

Jeżeli jednak nie podamy wartości dla wszystkich parametrów formalnych, wystąpi błąd:

```
>>> kto("Zdziś")

Traceback (most recent call last):
  File "<pyshell#17>", line 1, in -toplevel-
    kto("Zdziś")
```

```
TypeError: kto() takes exactly 3 arguments (1 given)
```

Możemy tego uniknąć, podając domyślne wartości argumentów. Zdefiniujmy funkcję od nowa:

```
>>> def kto(imie,wiek=18,jaki="oczytany"):
      print imie,"mimo swych",wiek,"lat jest bardzo",jaki+"m człowiekiem"
```

Teraz wystarczy jednak podać imię, by funkcja zadziałała:

```
>>> kto("Zdziś")
Zdziś mimo swych 18 lat jest bardzo oczytanym człowiekiem
>>> kto("Zdziś",44)
Zdziś mimo swych 44 lat jest bardzo oczytanym człowiekiem
>>> kto("Zdziś",22,"krnąbrny")
Zdziś mimo swych 22 lat jest bardzo krnąbrnym człowiekiem
```

Jedynym parametrem wymaganym pozostaje imię, gdyż nie podaliśmy dla niego wartości domyślnej:

```
>>> kto()

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in -toplevel-
    kto()
TypeError: kto() takes at least 1 argument (0 given)
```

Parametry do funkcji, możemy przekazywać albo podając ich wartości w kolejności podanej w nagłówku definicji funkcji, albo w dowolnej kolejności, wykorzystując ich nazwy:

```
>>> kto(imie="Jan",wiek=4,jaki="dostojny")
Jan mimo swych 4 lat jest bardzo dostojnym człowiekiem
>>> kto(jaki="dostojny",wiek=4,imie="Jan")
Jan mimo swych 4 lat jest bardzo dostojnym człowiekiem
```

Jeżeli parametry posiadają wartości domyślne, możemy przekazywać tylko wybrane z nich:

```
>>> kto(jaki="dostojny",imie="Jan")
Jan mimo swych 18 lat jest bardzo dostojnym człowiekiem
```

Parametry znajdujące się na początku listy i na właściwych sobie pozycjach, nie muszą mieć podanej nazwy:

```
>>> kto("Jan",jaki="dostojny")
Jan mimo swych 18 lat jest bardzo dostojnym człowiekiem
```

Parametry znajdujące się po parametrach wymienionych z nazwy, nawet na właściwych sobie pozycjach, muszą mieć podaną nazwę:

```
>>> kto(imie="Jan",4,"dostojny")
SyntaxError: non-keyword arg after keyword arg
```

8.6 Funkcje z nieznaną liczbą parametrów

Jeżeli w momencie definiowania funkcji nie jesteśmy w stanie określić liczby argumentów, które będą do niej przekazywane, poprzedzamy nazwę parametru formalnego oznaczającego wszystkie pozostałe argumenty funkcji gwiazdką:

```
>>> def suma(*arg):
      s=0
      for x in arg:
            s+=x
      return s
```

Teraz funkcja zadziała dla dowolnej liczby argumentów:

```
>>> suma()
0
>>> suma(1)
1
>>> suma(1,2)
3
>>> suma(1,2,3)
6
>>> suma(*range(10))
45
```

8.7 Funkcje rekurencyjne

Funkcje rekurencyjne to funkcje, które odwołują się do samych siebie. Dobrym przykładem funkcji rekurencyjnej jest silnia:

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

>>> silnia(1)
1
>>> silnia(2)
2
>>> silnia(5)
120
```

Wykorzystujemy tu fakt, że n!=(n-1)!*n.
Każda funkcja oparta na iteracji, może zostać przedstawiona w postaci rekurencyjnej. Np. suma:

```
>>> def suma(*n):
      if n:
            return n[0]+suma(*list(n)[1:])      # konwersja na listę, gdyż krotka nie
może być przycięta
      else:
            return 0


>>> suma(1)
1
>>> suma(1,2,3)
6
>>> suma(*range(100))
4950
```

Nie zawsze jednak funkcja w postaci rekurencyjnej jest jednak użyteczna. Przykładem nieefektywności rekursji jest funkcja wyliczająca n-ty element ciągu Fibonacciego (0,1,1,2,3,5,8,13,21,...).

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

Wypróbujmy:

```
>>> fib(1)
1
>>> fib(3)
2
>>> fib(8)
21
>>> fib(20)
6765
```

Na razie działa dobrze. Spróbujmy większej liczby n:

```
>>> fib(50)
```

Czekamy, czekamy, a wyniku jak nie było, tak nie ma. Najsensowniej będzie przerwać działanie funkcji wciskając kombinację klawiszy CTRL+C.
Dlaczego funkcja liczy tak powoli? Każde wywołanie funkcji powoduje jej ponowne dwukrotne wywołanie dla n>=2. A zatem, dla n=50, liczba wywołań funkcji wyniesie około 2^{49} razy. Nawet jeśli pojedyncze wywołanie funkcji zabiera tylko jedną dziesięciomilionową sekundy, to wykonanie 2^{49} wywołań zajmie komputerowi prawie dwa lata.

Tę samą funkcję da się przedstawić w szybkiej wersji iteracyjnej:

```
>>> def fib(n):
      if n<2:
            return n
      a, b = 0, 1             # 0 podstawiamy pod a, 1 pod b
      for x in range(1, n):   # potrzebujemy n-1 iteracji
            a, b = b, a+b     # b podstawiamy pod a, sumę pod b
      return b
```

Sprawdźmy:

```
>>> fib(0)
0
>>> fib(3)
2
>>> fib(8)
21
>>> fib(20)
6765
>>> fib(50)
12586269025L
```

8.8 Przykład: losowanie tekstu

Spróbujemy teraz napisać funkcję, która wygeneruje losowe zdanie zawierające podaną liczbę (domyślnie 5) losowo wygenerowanych wyrazów.

```
>>> def brednie(wyrazy=5):
      # funkcja generuje losowe zdanie o zadanej liczbie wyrazów
      from random import seed,randint
      seed()
      tekst=""
      for wyraz in range(wyrazy):
            for litera in range(randint(1,10)):            # między 1 a 10 liter w
wyrazie
                  tekst+=chr(randint(ord('a'),ord('z')))   # litery od a do z
            tekst+=" "
      return (tekst[:-1]+".").capitalize()                 # z dużej litery, a na końcu
kropka
```

Wypróbujmy:

```
>>> brednie()
'Phehwbbxjb gfhcgj uygnlabrog maicfvg xwi.'
>>> brednie()
'Wjzxo xqvgimdsh mvbr gwd lorcf.'
>>> brednie(1)
'Hhmgicsqnv.'
>>> brednie(4)
'Kxz adlokyjetg bwj wyranlsbrn.'
>>> brednie(9)
```

'Drdoy nekg lsnqdfysh cqgj qahuvjdqdk ijxjoxqtr khdjewr cbssjwo ftkdgeyal.'

8.9 Przykład: wyliczanie odległości między dwoma punktami na płaszczyźnie

Spróbujemy teraz napisać program do wyliczania odległości między dwoma punktami na płaszczyźnie. Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **punkty.py**.

```
# program wylicza odległość między dwoma punktami
from math import hypot
p1x,p1y = input("Podaj współrzędne poziomą i pionową pierwszego punktu >")
p2x,p2y = input("Podaj współrzędne poziomą i pionową drugiego punktu >")
print "Odległość między tymi punktami wynosi %.3f" % hypot(p1x-p2x,p1y-p2y)
```

Wypróbujmy:

```
>>> ================================ RESTART ================================
>>>
Podaj współrzędne poziomą i pionową pierwszego punktu >0,1
Podaj współrzędne poziomą i pionową drugiego punktu >1,0
Odległość między tymi punktami wynosi 1.414
>>> ================================ RESTART ================================
>>>
Podaj współrzędne poziomą i pionową pierwszego punktu >-1,-2
Podaj współrzędne poziomą i pionową drugiego punktu >3,1
Odległość między tymi punktami wynosi 5.000
>>>
```

8.10 Ćwiczenia kontrolne

I. Zdefiniuj funkcję "geo", która dla podanych trzech parametrów: n=numer elementu ciągu, a1=wartość pierwszego elementu ciągu (domyślnie 1), q=wartość iloczynu ciągu geometrycznego (domyślnie 2) zwróci n-ty element ciągu geometrycznego.

II. Zdefiniuj funkcję "avg", która dla dowolnej liczby parametrów zwróci ich średnią arytmetyczną (lub 0 dla 0 parametrów).

LEKCJA 9 – SŁOWNIKI

W trzech omówionych dotąd typach sekwencji – listach, krotkach i napisach – dostęp do dowolnego elementu możliwy był poprzez podanie jego indeksu. Odmiennego rodzaju typem złożonym jest słownik (ang. *dictionary*). W słowniku dostęp do dowolnej wartości przechowywanej w słowniku możliwy jest poprzez podanie klucza do niej. Słownik składa się zatem ze zbioru kluczy i zbioru wartości, gdzie każdemu kluczowi przypisana jest pojedyncza wartość. Zależność między kluczem a jego wartością nazywana bywa odwzorowaniem. Klucz nie musi być liczbą, tak jak jest nią indeks, wystarczy, że jest typu niezmiennego. Można powiedzieć więc, że o ile lista czy krotka odwzorowuje liczby całkowite (indeksy) na obiekty dowolnego typu, o tyle słownik odwzorowuje obiekty dowolnego typu niezmiennego na obiekty dowolnego typu.
W języku Python do tworzenia słowników używamy nawiasów klamrowych, np.:

```
>>> tel = {"policja":997, "straz":998, "pogotowie":999}
>>> tel
{'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Klucze nie muszą być tekstem, a wartości liczbami:

```
>>> bohater={"hans":"kloss","james":"bond"}
>>> ujemne={7:-7,3:-3}
```

Specyficzny słownik zwraca funkcja `vars`( ). Zawiera on wszystkie dostępne aktualnie zmienne:

```
>>> vars()
{'tel': {'policja': 997, 'pogotowie': 999, 'straz': 998}, 'bohater': {'hans': 'kloss',
'james': 'bond'}, '__builtins__': <module '__builtin__' (built-in)>, 'ujemne': {3: -3,
7: -7}, '__name__': '__main__', '__doc__': None}
```

Aby ustalić ilość kluczy pamiętanych w słowniku używamy funkcji `len`:

```
>>> len(tel)
3
```

Aby otrzymać wartość podanego klucza używamy nawiasów kwadratowych:

```
>>> tel ["policja"]
997
>>> bohater["hans"]
'kloss'
>>> ujemne[7]
-7
```

Zwróćmy uwagę, że `ujemne[7]` nie oznacza siódmego elementu słownika, lecz taki element słownika, którego klucz stanowi liczba 7.

Aby dopisać nowy klucz:

```
>>> tel["taxi"]=919
>>> tel
{'taxi': 919, 'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Aby zmodyfikować istniejącą wartość:

```
>>> tel["taxi"]=9622
>>> tel["taxi"]
9622
```

Zapis

```
>>> tel2=tel
```

nie skopiuje zawartości `tel` do `tel2` ale jedynie stworzy drugie odwołanie do tych wartości. A zatem operacja

```
>>> tel2["taxi"]=9666
```

zmieni zarówno wartości `tel` jak i `tel2`, gdyż są to te same dane pod dwoma różnymi nazwami:

```
>>> tel
{'taxi': 9666, 'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Aby skopiować zawartość jednego słownika do drugiego używamy metody `copy`:

```
>>> tel2=tel.copy()
```

W ten sposób otrzymaliśmy dwa różne słowniki, początkowo o tych samych wartościach.

```
>>> tel["taxi"]=9622
>>> tel
{'taxi': 9622, 'policja': 997, 'pogotowie': 999, 'straz': 998}
>>> tel2
{'taxi': 9666, 'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Aby usunąć wartość ze słownika używamy instrukcji `del`:

```
>>> del tel['taxi']
>>> tel
{'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Aby wyczyścić cały słownik używamy metody `clear`:

```
>>> tel.clear()
>>> tel
{}
```

Aby zaktualizować słownik w oparciu o inny słownik używamy metody `update`:

```
>>> tel.update(tel2)
>>> tel
{'taxi': 9666, 'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Aby odczytać i od razu usunąć określoną wartość ze słownika używamy metody `pop`:

```
>>> tel.pop('taxi')
9666
>>> tel
{'policja': 997, 'pogotowie': 999, 'straz': 998}
```

Aby odczytać i od razu usunąć nieokreśloną wartość ze słownika używamy metody `popitem`:

```
>>> tel.popitem()
('policja', 997)
>>> tel
{'pogotowie': 999, 'straz': 998}
```

Aby sprawdzić zawartość określonego klucza w słowniku używamy operatora `in`:

```
>>> 'taxi' in tel
False
>>> 'taxi' in tel2
True
```

Można też użyć w tym celu metody `has_key`:

```
>>> tel2.has_key("pogotowie")
True
```

Aby uzyskać listę kluczy występujących w słowniku, używamy metody `keys`:

```
>>> tel.keys()
['pogotowie', 'straz']
```

Aby uzyskać listę wartości występujących w słowniku, używamy metody `values`:

```
>>> tel.values()
[999, 998]
```

Można w ten sposób sprawdzić występowanie określonych wartości:

```
>>> 999 in tel.values()
True
```

Aby skonwertować słownik na napis:

```
>>> str(tel2)
"{'taxi': 9666, 'policja': 997, 'pogotowie': 999, 'straz': 998}"
```

Aby usunąć cały słownik:

```
>>> del tel
>>> tel

Traceback (most recent call last):
  File "<pyshell#35>", line 1, in -toplevel-
    tel
NameError: name 'tel' is not defined
```

Przykład 1. Napisz program „liczby_slownie.py”, który zamieni podaną przez użytkownika zapisaną słownie wartość (z zakresu od 1 do 59) na odpowiadającą jej liczbę dziesiętną.
Plan programu. Słowny zapis wartości składa się z sekwencji słów (np. trzydzieści trzy), z których każdemu przypisana jest pewna wartość. Wartość odpowiadającą całemu zapisowi otrzymuje się ze zsumowania wartości poszczególnych słów (np. trzydzieści (30) trzy (3) = 30 +3 = 33). Program powinien zatem rozbić wprowadzony przez użytkownika napis (`raw_input`) na wyrazy składowe (`split`), dla każdego wyrazu (`for x in l`) oznaczającego wartość (`if x in w`) ustalić odpowiadającą mu wartość (`w[x]`), a następnie dodać je do siebie (`s+=w[x]`).
Kod źródłowy:

```
def przelicz(n=""):
    w={"jeden":1, "dwa":2, "trzy":3, "cztery":4, "pięć":5, "sześć":6,
       "siedem":7, "osiem":8, "dziewięć":9, "dziesięć":10, "jedenaście":11,
       "dwanaście":12, "trzynaście":13, "czternaście":14, "piętnaście":15,
       "szesnaście":16, "siedemnaście":17, "osiemnaście":18,
       "dziewiętnaście":19, "dwadzieścia":20, "trzydzieści":30,
       "czterdzieści":40, "pięćdziesiąt":50}
    l=n.split()
    s=0
    for x in l:
       if x in w:
           s+=w[x]
    return s

t=raw_input("Wpisz liczbę od 1 do 59>")
print "Wartość:", przelicz(t)
```

Przykłady działania:

```
>>> ================================ RESTART ================================
>>>
Wpisz liczbę od 1 do >czterdzieści trzy
Wartość: 43
>>> ================================ RESTART ================================
```

```
>>>
Wpisz liczbę od 1 do 59>dwadzieścia siedem
Wartość: 27
>>> ================================ RESTART ================================
>>>
Wpisz liczbę od 1 do 59>bla bla dwa
Wartość: 2
```

Przykład 2. Napisz program „liczby_rzymskie.py”, który zamieni całkowitą liczbę dziesiętną na odpowiadającą jej liczbę rzymską.
Plan programu. Liczby rzymskie składają się z ciągu liter, z których każda oznacza pewną wartość (np. M – 1000, D – 500, C – 100, L – 50), a które dodane do siebie dadzą wartość reprezentowanej liczby. Liczbę zapisuje się możliwie najkrócej, czyli używa się liter o maksymalnej możliwej wartości. Litery oznaczające większe wartości umieszcza się przed tymi oznaczającymi mniejsze (czyli ML oznacza 1050), są jednak od tej reguły wyjątki, np. 900 zapisuje się jako CM, a nie DCCCC.
Gdy chcemy zamienić całkowitą liczbę dziesiętną na odpowiadającą jej liczbę rzymską musimy najpierw ustalić pierwszą literę liczby rzymskiej. Będzie nią największa wartość, która jest nie mniejsza od konwertowanej liczby. Kiedy tę wartość ustalimy, zapamiętujemy odpowiadającą jej literę, a samą wartość odejmujemy od konwertowanej liczby. Kiedy skonwertujemy ostatnią jedynkę (I), oznaczać to będzie, że liczba została w całości skonwertowana – można więc wyświetlić wynik.
Do zapamiętania wszystkich znanych wartości liter w liczbach rzymskich wykorzystamy słownik (`rzym`). Jego kluczami będą wartości liter, a wartościami odpowiadające im litery. Słownik obejmował będzie również wszystkie wyjątki.
Program zacznie działanie od wczytania liczby do skonwertowania (`x`). Następnie stworzymy listę wartości liter (`r`) uporządkowaną malejąco poprzez wydobycie listy kluczy ze słownika (`rzym.keys`), rosnące jej posortowanie (`r.sort`), a wreszcie odwrócenie kolejności (`r.reverse`). Ciąg zawierający wynik konwersji (`lr`) inicjalizujemy jako pusty.
Następnie, dla każdej wartości `i` z listy `r`, tak długo jak jest ona większa lub równa pozostałej do konwersji liczbie `x`, odpowiadającą jej sekwencję liter (`rzym[i]`) dołączamy do ciągu wynikowego (`lr`), a samą wartość odejmujemy (`x -= i`) od liczby pozostałej do konwersji.
Po zakończeniu powyższej pętli, wyświetlamy rezultat konwersji (`print lr`).
Kod źródłowy:

```
rzym = { 1000:"M", 900:"CM", 500:"D", 400:"CD", 100:"C", 90:"XC", 50:"L",
         40:"XL", 10:"X", 9:"IX", 5:"V", 4:"IV", 1:"I" }

x = input("Podaj liczbę całkowitą:")
print "Liczba",`x`,"w notacji rzymskiej to:",
r = rzym.keys()
r.sort()
r.reverse()
lr = ""
for i in r:
    while i <= x:
        lr += rzym[i]
        x -= i
print lr
```

Przykład uruchomienia (**F5**):

```
>>> ================================ RESTART ================================
>>>
Podaj liczbę całkowitą:13
Liczba 13 w notacji rzymskiej to: XIII
>>> ================================ RESTART ================================
>>>
Podaj liczbę całkowitą:1097
Liczba 1097 w notacji rzymskiej to: MXCVII
```

Ćwiczenia kontrolne

Ćwiczenie I. Napisz program „liczby_slownie2.py”, który dla wprowadzonej liczby dziesiętnej (z zakresu 1-1999) wyświetli jej wartość zapisaną słownie.

Ćwiczenie II. Napisz program „liczby_rzymskie2.py”, który przeliczy wprowadzoną liczbę rzymską na jej postać dziesiętną.

LEKCJA 10 – ZBIORY I REKORDY

Zbiory

Przejdźmy do trybu interaktywnego Pythona.
W Pythonie możemy tworzyć tak zbiory zmienne:

```
>>> A=set([1,2,3])
>>> A
set([1, 2, 3])
```

jak i niezmienne:

```
>>> B=frozenset([2,3,4])
>>> B
frozenset([2, 3, 4])
```

Aby stworzyć zbiór pusty napiszemy:

```
>>> C=set()
>>> C
set([])
```

Zbiory zmienne mogą być powiększane i zmniejszane:

```
>>> A.discard(2)
>>> A
set([1, 3])
>>> A.add(5)
>>> A
set([1, 3, 5])
```

Zbiory niezmienne nie mogą być ani zmniejszane:

```
>>> B.discard(2)

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in -toplevel-
    B.discard(2)
AttributeError: 'frozenset' object has no attribute 'discard'
```

ani powiększane:

```
>>> B.add(7)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in -toplevel-
    B.add(7)
AttributeError: 'frozenset' object has no attribute 'add'
>>>
```

Zbiory niezmienne mogą być kluczami w słownikach:

```
>>> d={B:7}
>>> d
{frozenset([2, 3, 4]): 7}
```

i elementami innych zbiorów:

```
>>> C.add(B)
>>> C
set([frozenset([2, 3, 4])])
```

Zbiory zmienne nie mogą być ani kluczami w słownikach:

```
>>> d={A:7}

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in -toplevel-
    d={A:7}
TypeError: set objects are unhashable
```

ani elementami innych zbiorów:

```
>>> C.add(A)

Traceback (most recent call last):
  File "<pyshell#17>", line 1, in -toplevel-
    C.add(A)
TypeError: set objects are unhashable
```

Operacje na zbiorach

Aby ustalić liczbę elementów zbioru piszemy:

```
>>> len(A)
3
>>> len(B)
3
>>> len(C)
1
```

Aby sprawdzić, czy dany obiekt jest elementem zbioru, piszemy:

```
>>> 2 in A
False
>>> 2 in B
True
>>> 5 in A
True
>>> 5 in B
False
```

Aby sprawdzić, czy dany obiekt nie jest elementem zbioru, piszemy:

```
>>> 3 not in A
False
>>> 3 not in B
False
>>> 7 not in A
True
>>> 7 not in B
True
```

Aby sprawdzić czy dany zbiór jest podzbiorem innego piszemy:

```
>>> set([1,3])<=A
True
>>> set([3,4])<=B
True
```

```
>>> set([5])<=A
True
>>> set([1,3,5])<=A
True
```

Lub:

```
>>> A.issubset(B)
False
```

Aby sprawdzić czy dany zbiór jest nadzbiorem innego piszemy:

```
>>> A>=set([1,3])
True
>>> B>=set([3,4])
True
```

Lub:

```
>>> A.issuperset(B)
False
```

Aby połączyć dwa zbiory piszemy:

```
>>> D=A | B
>>> D
set([1, 2, 3, 4, 5])
```

Aby określić część wspólną dwóch zbiorów piszemy:

```
>>> E=A & B
>>> E
set([3])
```

Aby określić różnicę dwóch zbiorów piszemy:

```
>>> A-B
set([1, 5])
>>> B-A
frozenset([2, 4])
```

(Typ zbioru wynikowego jest typem pierwszego ze zbiorów.)

Aby określić różnicę symetryczną dwóch zbiorów piszemy:

```
>>> F=A^B
>>> F
set([1, 2, 4, 5])
```

Przykład programu wykorzystującego zbiory

Wykorzystując zbiory rozwiążemy teraz ćwiczenie II z lekcji 7:
„Napisz program „lotto.py”, który wyświetli 6 losowych i nie powtarzających się liczb z zakresu od 1 do 49.”

Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **lotto.py**.

```
# program losuje 6 liczb od 1 do 49
from random import choice
```

```
Wylosowane = set()
while len(Wylosowane) < 6:
   Wylosowane.add(choice(range(1,50)))
for x in Wylosowane:
   print x,
```

Wypróbujmy (F5):

```
>>> ================================ RESTART ================================
>>>
1 36 8 22 24 29
```

Rekordy

Aby używać rekordów, musimy wpierw zdefiniować ich klasę.
W Pythonie lista pól w rekordzie może ulec zmianie w trakcie działania programu.
Dlatego możemy zdefiniować klasę jako pustą:

```
>>> class Adres:
        pass
```

A następnie utworzyć rekord w tej klasie:

```
>>> a=Adres( )
```

I dowolnie go rozszerzać:

```
>>> a.ulica="Matejki"
>>> a.numer="14a/2"
>>> a.kod="71-128"
>>> a.miasto="Szczecin"
```

Aby poznać zawartość poszczególnych pól rekordu piszemy:

```
>>> a.ulica
'Matejki'
>>> a.numer
'14a/2'
>>> a.kod
'71-128'
>>> a.miasto
'Szczecin'
```

Oczywiście, odpowiednie pola muszą istnieć:

```
>>> a.panstwo

Traceback (most recent call last):
  File "<pyshell#148>", line 1, in -toplevel-
    a.panstwo
AttributeError: Adres instance has no attribute 'panstwo'
```

Aby poznać zawartość wszystkich pól rekordu piszemy:

```
>>> a.__dict__
{'numer': '14a/2', 'miasto': 'Szczecin', 'ulica': 'Matejki', 'kod': '71-128'}
```

Możemy także określić listę pól już w momencie definiowania klasy, podając ich domyślne wartości:

```
>>> class Osoba:
        imie="Jan"
```

```
        nazwisko="Kowalski"
```

Aby stworzyć klasę opartą o inne klasy, podajemy je jako parametry:

```
>>> class Pracownik(Osoba, Adres):
        pensja=1000.00
```

Otrzymana klasa Pracownik będzie miała wszystkie pola klas Osoba i Adres, a dodatkowo pole pensja. Sprawdźmy:

```
>>> p=Pracownik()
>>> p2=Pracownik()
>>> p2.imie="Adam"
>>> p2.ulica="Torfowa"
>>> p2.pensja=700
>>> p.imie
'Jan'
>>> p.ulica

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    p.ulica
AttributeError: Pracownik instance has no attribute 'ulica'

>>> p.pensja
1000.0
>>> p2.imie
'Adam'
>>> p2.ulica
'Torfowa'
>>> p2.pensja
700
```

Trzeba przy tym pamiętać, że __dict__ zwraca słownik tylko tych pól, których wartości różnią się od domyślnych:

```
>>> p2.__dict__
{'imie': 'Adam', 'pensja': 700, 'ulica': 'Torfowa'}
>>> p.__dict__
{ }
```

Przykład – lista ocen studentów

Spróbujemy teraz napisać program, który posłuży do przechowania listy ocen studentów z egzaminu.
Aby przejść do edycji nowego programu należy z menu **File** wybrać polecenie **New Window.** Otworzy się nowe okno, przeznaczone do edycji programu. Wybierzmy z menu **File** polecenie **Save As**. Wybieramy folder Moje dokumenty, następnie wpisujemy nazwę **studenci.py**.

```
# Lista Ocen Studentów
class Student:
   imie=""
   nazwisko=""
   ocena=0.0

studenci = []
nr = 0
while True:
   naz=raw_input('Podaj nazwisko studenta nr %i (ENTER=koniec): ' % (nr+1))
   if not naz: break
   studenci.append(Student())
   studenci[nr].nazwisko = naz
   studenci[nr].imie = raw_input('Podaj imię studenta nr %i > ' % (nr+1))
   studenci[nr].ocena = float(raw_input('Podaj ocenę studenta nr %i > ' % (nr+1)))
   nr += 1
```

```
print
print "%-4s %-14s %-10s %7s" % ("L.p.","Nazwisko","Imię","Ocena")
for s in studenci:
   print "%3i. %-14s %-10s %7.1f" % (studenci.index(s)+1,s.nazwisko, s.imie, s.ocena)
```

Wypróbujmy (F5):

```
>>> ================================ RESTART ================================
>>>
Podaj nazwisko studenta nr 1 (ENTER=koniec): Adamczak
Podaj imię studenta nr 1 > Piotr
Podaj ocenę studenta nr 1 > 3.5
Podaj nazwisko studenta nr 2 (ENTER=koniec): Gierek
Podaj imię studenta nr 2 > Franz
Podaj ocenę studenta nr 2 > 2.0
Podaj nazwisko studenta nr 3 (ENTER=koniec): Hubert
Podaj imię studenta nr 3 > Anna
Podaj ocenę studenta nr 3 > 4.5
Podaj nazwisko studenta nr 4 (ENTER=koniec):

L.p. Nazwisko      Imię       Ocena
  1. Adamczak      Piotr        3.5
  2. Gierek        Franz        2.0
  3. Hubert        Anna         4.5
```

Ćwiczenia kontrolne

I. Zmień program „lotto” w taki sposób, by wyświetlał wylosowane liczby od najmniejszej do największej.

II. Zmień program „lista studentów” w taki sposób, by dodatkowo obliczał średnią ocen wszystkich studentów

LEKCJA 11 - PRZETWARZANIE LIST

Programy bardzo często przetwarzają sekwencje danych. Jakkolwiek szczegóły tego przetwarzania bywają rozmaite, istnieje kilka podstawowych typów operacji wykonywanych na sekwencjach danych. Zostaną one przedstawione poniżej, wraz z informacją, jak wygodnie zrealizować je w języku Python.
Zaczniemy od szybkiego generowania list. Wiemy już, jak tworzyć listy zawierające wiele takich samych elementów:

```
>>> l=[0]*20
>>> l
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

lub powtarzający się układ:

```
>>> l=[3,5,2]*6
>>> l
[3, 5, 2, 3, 5, 2, 3, 5, 2, 3, 5, 2, 3, 5, 2, 3, 5, 2]
```

oraz jak tworzyć listy, których elementy należą do ciągu arytmetycznego:

```
>>> l=range(1,21,2)
>>> l
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Poznamy teraz wygodne narzędzie do szybkiego tworzenia list o nawet bardzo złożonej zawartości, w oparciu o inną listę. Tym narzędziem są wytworniki list (ang. *list comprehensions*). Wytworniki dostępne są w pięciu postaciach:

- prostej,
- prostej warunkowej,
- rozszerzonej,
- rozszerzonej z jednym warunkiem,
- rozszerzonej z wieloma warunkami.

Postać prosta wytwornika ma następującą składnię:
[*wyrażenie* for *zmienna* in *sekwencja*] i daje w wyniku listę, zawierającą wartości wyrażenia obliczone dla elementów sekwencji wejściowej. Przykłady:

- Podwojenie wartości:

```
>>> [2*x for x in l]
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38]
```

- Stworzenie par (*x*, kwadrat *x*):

```
>>> [(x, x*x) for x in range(1,5)]
[(1, 1), (2, 4), (3, 9), (4, 16)]
```

- Tabela kodowa ASCII:

```
>>> [(x, ord(x)) for x in "ABCDEF"]
[('A', 65), ('B', 66), ('C', 67), ('D', 68), ('E', 69), ('F', 70)]
```

- Lista zawierająca 10 pustych list:

```
>>> [ [] for x in range(10) ]
[[], [], [], [], [], [], [], [], [], []]
```

Postać prosta warunkowa pozwala umieszczać na liście tylko takie elementy, które spełniają pewien warunek (operację usuwania z listy elementów niespełniających określonego warunku nazywamy

filtrowaniem danych). Ma następującą składnię:
[*wyrażenie* for *zmienna* in *sekwencja* if *warunek*]

Przykłady:

- Tylko liczby większe od 10:

```
>>> [x for x in l if x>10]
[11, 13, 15, 17, 19]
```

- Tylko liczby podzielne przez 3 lub 5:

```
>>> [x for x in range(1,20) if not (x%3) or not (x%5)]
[3, 5, 6, 9, 10, 12, 15, 18]
```

- Tabela kodowa ASCII tylko dla samogłosek:

```
>>> [(x, ord(x)) for x in "ABCDEF" if x in "AEIOUY"]
[('A', 65), ('E', 69)]
```

Postać rozszerzona pozwala tworzyć nową listę w oparciu o więcej niż jedną istniejącą listę; ma następującą składnię:
[*wyrażenie* for *zmienna1* in *sekwencja1*
for *zmienna2* in *sekwencja2*
...]
Przykłady:

- Pary każdy element z każdym:

```
>>> [(x,y) for x in range(1,5)
 for y in range(4,0,-1)]
[(1, 4), (1, 3), (1, 2), (1, 1), (2, 4), (2, 3), (2, 2), (2, 1), (3, 4), (3, 3), (3, 2), (3, 1), (4, 4), (4, 3), (4, 2), (4, 1)]
```

- Różnica między wartością z pierwszej i drugiej listy:

```
>>> [x-y for x in range(1,5)
 for y in range(4,0,-1)]
[-3, -2, -1, 0, -2, -1, 0, 1, -1, 0, 1, 2, 0, 1, 2, 3]
```

- Sklejenie napisu z wartości pochodzących z trzech list:

```
>>> [`x`+y+`z` for x in [1,2]
 for y in ['A','B']
 for z in [0,3] ]
['1A0', '1A3', '1B0', '1B3', '2A0', '2A3', '2B0', '2B3']
```

Postać rozszerzona z jednym warunkiem pozwala na określenie pojedynczego warunku, który muszą spełniać dane kwalifikujące się na listę wynikową. Jej składnia jest następująca:
[*wyrażenie* for *zmienna1* in *sekwencja1*
for *zmienna2* in *sekwencja2*
...
if *warunek*]
Przykłady:

- Pary każdy element z każdym, tylko jeżeli pierwszy element jest mniejszy od drugiego:

```
>>> [(x,y) for x in range(1,5)
 for y in range (6,3,-1)
 if x<y]
[(1, 6), (1, 5), (1, 4), (2, 6), (2, 5), (2, 4), (3, 6), (3, 5), (3, 4), (4, 6), (4, 5)]
```

- Pary każdy element z każdym, tylko jeżeli suma elementów jest mniejsza od 7:

```
>>> [(x,y) for x in range(1,5)
 for y in range (6,3,-1)
 if x+y<7]
[(1, 5), (1, 4), (2, 4)]
```

- Pary każdy element z każdym, pod warunkiem, że pierwszy element jest parzysty, lub drugi jest nieparzysty:

```
>>> [(x,y) for x in range(1,5)
 for y in range (6,2,-1)
 if not (x%2) or y%2]
[(1, 5), (1, 3), (2, 6), (2, 5), (2, 4), (2, 3), (3, 5), (3, 3), (4, 6), (4, 5), (4, 4), (4, 3)]
```

Postać rozszerzona z wieloma warunkami pozwala na określenie warunków, które muszą spełniać dane pobierane z poszczególnych list źródłowych. Jej składnia jest następująca:
[*wyrażenie*
for *zmienna1* in *sekwencja1* if *warunek1*
for *zmienna2* in *sekwencja2* if *warunek2*
...]
Przykład:

- Pary każdy element z każdym, pod warunkiem, że pierwszy element jest parzysty a drugi jest nieparzysty (z pierwszej listy bierzemy tylko elementy parzyste, a z drugiej – nieparzyste):

```
>>> [(x,y) for x in range(1,5) if not (x%2)
 for y in range (6,2,-1) if y%2]
[(2, 5), (2, 3), (4, 5), (4, 3)]
```

W Pythonie dostępnych jest pięć funkcji, których przeznaczeniem jest ułatwienie przetwarzania sekwencji danych. Pierwszą z nich jest funkcja apply, której działanie polega na wywołaniu funkcji z parametrami uzyskanymi z rozpakowania sekwencji (jest zatem identyczne z poprzedzeniem nazwy sekwencji gwiazdką, jak zostało to omówione w podrozdziale 5.2). Przykład użycia:

```
>>> dziel=lambda x,y,z: (x+y)/z
>>> dziel(3,5,2)
4
>>> xyz=(3,5,2)
>>> apply(dziel,xyz)
4
```

Funkcja map działa inaczej: pozwala wywołać określoną funkcję dla każdego elementu sekwencji z osobna. Zwraca listę rezultatów funkcji, o takiej samej długości jak listy parametrów. Przykłady użycia:

```
>>> map(lambda x: x*x, range(5))
[0, 1, 4, 9, 16]
>>> map(dziel, range(5), range(5), [2]*5)
[0, 1, 2, 3, 4]
```

W pierwszym z powyższych przykładów użyto funkcji jednoparametrowej (kwadratu), pobierając parametry z pojedynczej listy. W drugim użyto funkcji trójparametrowej (zdefiniowana wcześniej dziel), jej pierwszy parametr pobierany jest z pierwszej listy, drugi – z drugiej, a trzeci – z trzeciej.
Funkcja zip służy do **konsolidacji danych**, tj. operacji łączenia kilku list w jedną, w której wartość pojedynczego elementu listy wynikowej zależy od wartości pojedynczych elementów list źródłowych. Funkcja zip przyjmuje jako swoje parametry jedną lub więcej sekwencji, po czym zwraca listę krotek, których poszczególne elementy pochodzą z poszczególnych sekwencji.

```
>>> zip("abcdef",[1,2,3,4,5,6])
```

```
[('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5), ('f', 6)]
>>> zip(range(1,10),range(9,0,-1))
[(1, 9), (2, 8), (3, 7), (4, 6), (5, 5), (6, 4), (7, 3), (8, 2), (9, 1)]
```

W przypadku, gdy długości sekwencji są różne, wynikowa sekwencja jest skracana do najkrótszej spośród nich:

```
>>> zip("zip",range(0,9),zip(range(0,9)))
[('z', 0, (0,)), ('i', 1, (1,)), ('p', 2, (2,))]
```

Funkcja filter służy do filtrowania danych. Przyjmuje jako parametry funkcję oraz sekwencję, po czym zwraca sekwencję zawierającą te elementy sekwencji wejściowej, dla których funkcja zwróciła wartość logiczną True.
Tak na przykład filtruje się samogłoski:

```
>>> samogloska=lambda x: x.lower() in 'aeiou'
>>> samogloska('A')
True
>>> samogloska('z')
False
>>> filter(samogloska,"Ala ma kota, kot ma Ale")
'AaaoaoaAe'
```

Tak wszystkie inne znaki:

```
>>> filter(lambda x: not samogloska(x),"Ala ma kota, kot ma Ale")
'l m kt, kt m l'
```

A tak liczby parzyste:

```
>>> filter(lambda x: x%2-1, range(0,11))
[0, 2, 4, 6, 8, 10]
```

Funkcja reduce służy do **agregowania danych**, tj. operacji obliczenia pojedynczego wyrażenia, zależnego od wszystkich elementów listy źródłowej. Funkcja reduce przyjmuje jako parametry funkcję oraz sekwencję, zwraca pojedynczą wartość.
Na początek wykonuje funkcję dla dwóch pierwszych elementów sekwencji, następnie wykonuje funkcję dla otrzymanego w pierwszym kroku rezultatu i trzeciego elementu sekwencji, następnie wykonuje funkcję dla otrzymanego w drugim kroku rezultatu i czwartego elementu sekwencji, itd., aż dojdzie do końca sekwencji.
Np.: suma elementów:

```
>>> reduce(lambda x,y: x+y, [1,2,3])
6
```

Np.: iloczyn elementów:

```
>>> reduce(lambda x,y: x*y, [1,2,3,4])
24
```

Np.: suma kwadratów elementów:

```
>>> reduce(lambda x,y: x+y, map(lambda x: x*x, range(1,10)))
285
```

Na poniższym przykładzie zobaczymy, jak łączyć stosowanie poznanych tu konstrukcji.
Przykład. Dane są cztery listy: r1, r2, r3 i r4 liczące po 12 elementów. Zawierają one wartości miesięcznej sprzedaży przez kolejne cztery lata w pewnej firmie. Oblicz:

1. Sumę sprzedaży dla poszczególnych lat
2. Sumę sprzedaży w poszczególnych miesiącach przez cztery lata
3. Średnią sprzedaż miesięczną przez cztery lata

Rozwiązanie. Na początek przygotujemy losowe dane wejściowe.

```
>>> from random import *
>>> seed(2006)
>>> r1=[randint(20,50) for i in range(12)]
>>> r2=[randint(20,50) for i in range(12)]
>>> r3=[randint(20,50) for i in range(12)]
>>> r4=[randint(20,50) for i in range(12)]
>>> r1;r2;r3;r4
[37, 22, 48, 30, 37, 22, 27, 33, 25, 37, 44, 26]
[41, 26, 29, 31, 28, 34, 32, 44, 25, 40, 43, 40]
[38, 26, 23, 23, 34, 36, 32, 38, 36, 48, 40, 34]
[38, 39, 34, 48, 26, 50, 34, 33, 50, 20, 48, 41]
```

Ad. 1. Aby wyliczyć sumę sprzedaży dla poszczególnych lat posłużymy się funkcją reduce:

```
>>> reduce(lambda x,y:x+y,r1)
388
```

W powyższy sposób zagregowaliśmy listę elementów r1 do ich sumy. Aby wyliczyć sumę sprzedaży dla wszystkich lat, posłużymy się dodatkowo funkcją map:

```
>>> map(reduce,[lambda x,y:x+y]*4,[r1,r2,r3,r4])
[388, 413, 408, 461]
```

Wywoła ona funkcję reduce każdorazowo dla funkcji sumującej (zwróćmy uwagę w jaki sposób ją powieliliśmy, by odpowiadała liczbie lat) i danych z kolejnego roku.
Ad. 2. Aby zestawić ze sobą dane o sprzedaży w poszczególnych miesiącach dla wszystkich lat użyjemy funkcji zip:

```
>>> zip(r1,r2,r3,r4)
[(37, 41, 38, 38), (22, 26, 26, 39), (48, 29, 23, 34), (30, 31, 23, 48), (37, 28, 34, 26), (22, 34, 36, 50), (27, 32, 32, 34),
(33, 44, 38, 33), (25, 25, 36, 50), (37, 40, 48, 20), (44, 43, 40, 48), (26, 40, 34, 41)]
```

Aby wyliczyć sumę sprzedaży miesięcznej dla wszystkich lat, posłużymy się znaną już nam kombinacją funkcji reduce i map:

```
>>> map(reduce,[lambda x,y:x+y]*12, zip(r1,r2,r3,r4))
[154, 113, 134, 132, 125, 142, 125, 148, 136, 145, 175, 141]
```

Ad. 3. Aby wyliczyć średnią sprzedaż miesięczną przez cztery lata, posłużymy się ponownie funkcją map:

```
>>> map(lambda x: x/4.0, map(reduce,[lambda x,y:x+y]*12, zip(r1,r2,r3,r4)))
[38.5, 28.25, 33.5, 33.0, 31.25, 35.5, 31.25, 37.0, 34.0, 36.25, 43.75, 35.25]
```

Jedną z częściej wykonywanych na listach operacji jest ich porządkowanie (sortowanie i odwracanie kolejności). Przypomnijmy metodę list sort:

```
>>> l=[3,2,5,7]
>>> l.sort()
>>> l
[2, 3, 5, 7]
```

Jak widać, standardowo porządkuje ona elementy od najmniejszego do największego. Metoda sort posiada jednak parametr reverse, ustawienie którego wartości na True pozwala zmienić porządek sortowania:

```
>>> l=[3,2,5,7]
>>> l.sort(reverse=True)
>>> l
[7, 5, 3, 2]
```

W pewnych przypadkach, problem, który mamy z sortowaniem jest znacznie bardziej skomplikowany niż zmiana kolejności. Załóżmy, że mamy listę wyrazów, którą chcielibyśmy posortować alfabetycznie:

```
>>> L=["Ala","Ola","pies","dziadek","Tola","smyk"]
>>> L.sort()
>>> L
['Ala', 'Ola', 'Tola', 'dziadek', 'pies', 'smyk']
```

Jak widać, standardowe sortowanie nie jest poprawne, gdyż umieszcza wszystkie wyrazy pisane z dużej litery przed tymi pisanymi z małej. Może w tym pomóc parametr key, pozwalający określić funkcję, która skonwertuje dane do postaci zapewniającej prawidłowy rezultat porównania. W tym przypadku chodzi o sprowadzenie wszystkich wyrazów do małych liter – użyjemy zatem metody lower klasy str (jak widać, metody możemy wywoływać łącząc ich nazwy nie tylko z nazwami konkretnych obiektów, ale także z nazwą samej klasy – sam obiekt przekazany zaś zostanie jako pierwszy parametr metody):

```
>>> L=["Ala","Ola","pies","dziadek","Tola","smyk"]
>>> L.sort(key=str.lower)
>>>
>>> L
['Ala', 'dziadek', 'Ola', 'pies', 'smyk', 'Tola']
```

Inny przykład wykorzystania parametru key to sortowanie liczb zapisanych jako napisy. Domyślnie są one sortowane jako ciągi cyfr, co daje porządek leksykograficzny, a nie odpowiadający wartościom liczb:

```
>>> L=["11","2","20","7","55"]
>>> L.sort()
>>> L
['11', '2', '20', '55', '7']
```

Aby uzyskać sortowanie według wartości, wystarczy skonwertować napisy na liczby używając funkcji int:

```
>>> L=["11","2","20","7","55"]
>>> L.sort(key=int)
>>> L
['2', '7', '11', '20', '55']
```

Zauważmy, że parametr key jest w istocie funkcją (funkcje są zatem specyficznym typem danych, tzw. typem proceduralnym). Taką sytuację (wywołanie z funkcji innej funkcji, przekazanej do niej jako parametr) nazywamy wywołaniem zwrotnym (ang. *callback*).

Parametryzacja metody sort idzie jeszcze dalej, pozwalając podmienić całą funkcję wykonującą porównanie. Domyślnie jest to funkcja cmp; podmieniona funkcja musi tak samo jak cmp zwracać 0 dla dwóch parametrów równych sobie oraz 1 lub -1 w przypadku gdy zachodzi nierówność.

```
>>> cmp(1,2)
-1
>>> cmp(2,2)
0
>>> cmp(3,2)
1
```

Sortowanie liczb zapisanych jako napisy mogłoby wyglądać więc tak:

```
>>> def porownaj(x,y):
   return cmp(int(x), int(y))

>>> L=["11","2","20","7","55"]
>>> L.sort(cmp=porownaj)
>>> L
['2', '7', '11', '20', '55']
```

Funkcję porownaj zdefiniowaliśmy po to, by wykorzystać ją w jednym tylko miejscu. Python pozwala uniknąć definicji takich funkcji jednorazowego użytku dzięki wyrażeniu lambda. Wyrażenie to ma postać lambda *argumenty*: *wyrażenie*, a jego rezultatem jest anonimowa funkcja (którą można od razu wykonać, przekazać jako parametr, lub zapamiętać w zmiennej – tym samym nadając jej nazwę).
Zamiast definiować funkcję porownaj moglibyśmy napisać:

```
>>> L=["11","2","20","7","55"]
>>> L.sort(cmp=lambda x,y: cmp(int(x), int(y)))
>>> L
['2', '7', '11', '20', '55']
```

Wyrażeń lambda można oczywiście używać w każdej sytuacji, w której moglibyśmy użyć na ich miejscu funkcji, np.:

```
>>> print "%.2f" % (lambda x,y: x**y)(2,0.5)
1.41
```

Podstawową wadą wyrażeń lambda to niemożność wykorzystania w nich instrukcji nie będących wyrażeniami, np. print, if, for, while, itp. – choć istnieją sprytne sposoby na obejście tego problemu. Jakkolwiek wyrażenia lambda bywają wygodne, nie należy ich nadużywać, bo prowadzi to do nieprzejrzystego kodu źródłowego.
Jeszcze jeden przykład na porównywanie, tym razem sekwencji:

```
>>> L=[ ("Adam",15), ("Bogdan",19), ("Ala",17), ("Zenobia", 14) ]
>>> L.sort()
>>> L
[('Adam', 15), ('Ala', 17), ('Bogdan', 19), ('Zenobia', 14)]
```

Jak pamiętamy, porównywanie sekwencji polega na porównaniu między sobą pierwszych ich elementów, tylko gdy są równe – drugich, itd. Załóżmy, że powyższą listę chcielibyśmy posortować według lat, nie imion. Można to zrobić następująco (parametr cmp jest pierwszym parametrem metody sort, jego wartość może więc być podawana bez nazwy):

```
>>> L.sort(lambda x,y: cmp(x[1],y[1]))
>>> L
[('Zenobia', 14), ('Adam', 15), ('Ala', 17), ('Bogdan', 19)]
```

Jak widać, elementy listy są teraz ułożone według rosnącego wieku (drugiego elementu krotki).

Ćwiczenie 1. Używając wytwornika zbuduj listę zawierającą wszystkie liczby podzielne przez 3 z zakresu od 1 do 33. Następnie:

- Używając funkcji filter usuń z niej wszystkie liczby parzyste
- Używając wyrażenia lambda i funkcji map podnieś wszystkie elementy tak otrzymanej listy do sześcianu
- Odpowiednio używając funkcji reduce i len oblicz średnią arytmetyczną z elementów tak otrzymanej listy.

Ćwiczenie 2. Stwórz trzy listy zawierające po 5 elementów: *nazwiska* – z nazwiskami pracowników, *godziny* – z liczbą przepracowanych godzin, *stawka* – ze stawką w złotych za godzinę pracy. Wykorzystując funkcje zip, map, reduce i filter (oraz, ewentualnie, wytworniki list) wyświetl nazwiska i wypłaty (iloczyn

stawki godzinowej i liczby przepracowanych godzin) tych pracowników, którzy zarobili więcej, niż wyniosła średnia wypłata.

LEKCJA 12 – PLIKI

Do tej pory zajmowaliśmy się wyłącznie przechowywaniem danych w pamięci operacyjnej komputera. Pamięć operacyjna komputera jest jednak ulotna. Aby zabezpieczyć dane należy zapisać je w pliku dyskowym.

Jeżeli w trybie interaktywnym Pythona wpiszemy:

```
>>> f1 = open("plik1.txt","wb")
```

– spowodujemy stworzenie i otwarcie do zapisu pliku "plik1.txt" w aktualnym katalogu dyskowym (po uruchomieniu IDLE'a jest to katalog, w którym zainstalowano Pythona, domyślnie „C:\Python24". Choć zapisywać będziemy w pliku sam tekst, używamy trybu binarnego, w celu uniknięcia problemu z ustalaniem bieżącej pozycji w pliku.

Obiekty plikowe mają trzy podstawowe atrybuty:

- `name` zawiera nazwę pliku (tak jak podano ją przy otwarciu)

```
>>> f1.name
'plik1.txt'
```

- `mode` określa tryb, w jakim otwarto plik

```
>>> f1.mode
'w'
```

- `closed` określa czy plik jest zamknięty

```
>>> f1.closed
False
```

Pliki obsługujemy przy użyciu następujących metod:

- `write` zapisuje do pliku napis

```
>>> f1.write("Początek pliku")
```

Możemy teraz, korzystając z *Eksploratora Windows* przejść do katalogu, w którym znajduje się plik (przypominam, standardowo ścieżką dostępu będzie „C:\Python24\plik1.txt") i otworzyć go w *Notatniku*. Plik powinien znajdować się na swoim miejscu, jednak prawdopodobnie będzie pusty. Winne oczywiście jest buforowanie.

- `flush` zapisuje dane z bufora do pliku

```
>>> f1.flush()
```

Jeżeli teraz ponownie otworzymy „plik1.txt" w *Notatniku*, powinniśmy zobaczyć następującą zawartość:

```
Początek pliku
```

Metoda write nie kończy zapisanych danych znakiem końca linii. By przejść do kolejnej linii, musimy sami zapisać taki znak (`\n`):

```
>>> f1.write("\nDruga linia")
```

- `close` zapisuje dane z bufora do pliku i zamyka plik

```
>>> f1.close()
```

Normalne zakończenie programu powoduje automatyczne zamknięcie wszystkich otwartych plików, jednak programiści powinni samodzielnie zamykać wszystkie pliki, które otwarli.
Jeżeli teraz ponownie otworzymy „plik1.txt" w *Notatniku* (oczywiście poprzednią wersję, jeżeli nadal jest otwarta, należy już zamknąć, gdyż *Notatnik* nie aktualizuje zawartości edytowanego dokumentu po jego otwarciu z dysku), powinniśmy zobaczyć następującą zawartość:

```
Początek pliku
Druga linia
```

Spróbujemy teraz otworzyć zapisany przed chwilą plik do modyfikacji:

```
f1 = open("plik1.txt","r+b")
```

- `read` odczytuje z pliku napis

```
>>> print f1.read()
Początek pliku
Druga linia
```

- `tell` podaje aktualną pozycję w pliku

```
>>> f1.tell()
26L
```

- `seek` ustawia pozycję w pliku na podaną

```
>>> f1.seek(0)
```

Modyfikujemy zawartość pliku zapisując na istniejącej wcześniej pozycji:

```
>>> f1.write("Pierwsza linia")
```

Aby przesunąć pozycję pliku nie na pozycję wyrażoną absolutnie (od początku pliku), lecz względnie (od aktualnej pozycji), jako drugi parametr podajemy 1:

```
>>> f1.seek(-14,1)
```

Możemy wczytywać tylko fragment zawartości pliku o określonej długości:

```
>>> print f1.read(14)
Pierwsza linia
```

Aby przesunąć pozycję pliku względem jego końca, jako drugi parametr podajemy 2:

```
>>> f1.seek(0,2)
```

- `writelines` zapisuje do pliku sekwencję napisów (nie dodając automatycznie separatorów):

```
>>> f1.writelines(["\n3 linia","\n4 linia","\n5 linia"])
```

- `readlines` wczytuje z pliku sekwencję napisów:

```
>>> f1.seek(0)
>>> a=f1.readlines()
>>> print a
['Pierwsza linia\n', 'Druga linia\n', '3 linia\n', '4 linia\n', '5 linia']
```

Zauważmy, że `readlines` wykorzystuje znaki końca linii do podziału tekstu na napisy, nie usuwa ich jednak automatycznie.

- `truncate` skraca plik na podanej pozycji:

```
>>> f1.truncate(26)
>>> f1.seek(0)
>>> print f1.read()
Pierwsza linia
Druga linia
```

- `isatty` – zwraca True, jeżeli plik jest dołączony do urządzenia terminalowego:

```
>>> f1.isatty()
False
```

Przykładami takich plików są `sys.stdout` i `sys.stdin` (pamiętajmy przy tym, że są to strumienie kierujące dane z/do konsoli, stąd nie wszystkie operacje są dla nich dostępne):

```
>>> import sys
>>> sys.stdout.isatty()
True
```

Należy pamiętać, że użytkownik uruchamiając program może przekierować jego wejście i wyjście z poziomu systemu operacyjnego. Można to sprawdzić właśnie dzięki metodzie `isatty` – jeżeli dla któregoś z podanych plików zwraca ona wartość `False`, wejście lub wyjście zostało przekierowane do zwykłego pliku. Atrybuty `sys.stdout` i `sys.stdin` są zmiennymi, można więc zmienić je na dowolny inny plik, przekierowując w ten sposób wyjście/wejście wewnątrz programu:

```
>>> import sys
>>> ekran=sys.stdout
>>> sys.stdout = open("wyjscie.txt","w")
>>> print "Cokolwiek"
>>> print "Gdzie to się wyświetliło?"
>>> sys.stdout=ekran
>>> print open("wyjscie.txt","r").read()
Cokolwiek
Gdzie to się wyświetliło?
```

Co więcej, obiekty plikowe w Pythonie wcale nie muszą być połączone z jakimikolwiek plikami fizycznymi. Wystarczy tylko, że należą do klasy posiadającej używane metody. Można to wykorzystać np. do formatowania danych wyświetlanych instrukcją `print`:

```
>>> class centrowanie:
      def write(self, s):
         ekran.write(s.center(60))

>>> sys.stdout=centrowanie()
>>> print "Cokolwiek"
                         Cokolwiek

>>> print "O! Jak ładnie"
                       O! Jak ładnie

>>> sys.stdout=ekran
>>> print "Już normalnie!"
Już normalnie!
```

Przykład 1. Program, który kopiuje plik o nazwie podanej przez użytkownika, wydłużając nazwę nowego pliku o prefiks „kopia”.
Rozwiązanie z komentarzem:

```
n=raw_input("Podaj pełną nazwę pliku do skopiowania>")# n - nazwa oryginału
oryg = file(n,"rb")        # otwieramy oryginał do odczytu
kop = file("kopia "+n,"wb")# otwieramy kopię do zapisu
while True:                # powtarzamy
      b = oryg.read(1)     # wczytujemy 1 bajt z oryginału
```

```
      if not b: break      # nic się nie wczytało? Koniec pliku!
      kop.write(b)         # zapisujemy 1 bajt do kopii
kop.close()                # zamykamy kopię
oryg.close()               # zamykamy oryginał
print "Kopiowanie zakończone pomyślnie" # komunikat końcowy
```

Przykład uruchomienia:

```
>>> ================================ RESTART ================================
>>>
Podaj pełną nazwę pliku do skopiowania>plik1.txt
Kopiowanie zakończone pomyślnie
```

Przykład 2. Program „cezar.py”, który szyfruje wskazany przez użytkownika plik tekstowy w oparciu o szyfr Cezara (przesuwanie liter w alfabecie o podaną wartość).
Rozwiązanie z komentarzem:

```
t=raw_input("Podaj pełną nazwę pliku >")
p=input("Podaj przesunięcie >”)
f1 = open(t,"r+b")      # plik otwarty do modyfikacji
s=f1.read()             # wczytujemy do s tekst źródłowy
sz=""                   # sz oznacza tekst zaszyfrowany
for c in s:             # dla każdego znaku w s
    a=ord(c)            # wyliczamy kod ASCII
    if a > 32:          # znaków białych i sterujących nie szyfrujemy
        c=chr((a+p) % 256)    # inne przesuwamy
    sz+=c               # dodajemy do sz
f1.seek(0)              # ustawiamy pozycję pliku na jego początku
f1.write(sz)            # zapisujmy sz
f1.close()              # zamykamy plik
```

Wypróbujmy:

```
>>> ================================ RESTART ================================
>>>
Podaj pełną nazwę pliku >plik1.txt
Podaj przesunięcie >3
```

Przyjrzyjmy się teraz (w notatniku) zawartości pliku `plik1.txt` po zaszyfrowaniu:

```
Slhuzv}d olqld
Guxjd olqld
```

Aby rozszyfrować plik należy jeszcze raz uruchomić program „cezar.py”, i podać jako przesunięcie ujemną wartość liczby podanej przy szyfrowaniu:

```
>>> ================================ RESTART ================================
>>>
Podaj pełną nazwę pliku >plik1.txt
Podaj przesunięcie >-3
```

Przyjrzyjmy się teraz zawartości pliku `plik1.txt` po odszyfrowaniu:

```
Pierwsza linia
Druga linia
```

Ćwiczenie I. Napisz program „type.py”, który wyświetli na ekranie zawartość pliku o nazwie podanej przez użytkownika.
Ćwiczenie II. Napisz program „lista.py”, który:

a. a. Jeżeli na dysku nie ma pliku „lista.txt”, wczyta od użytkownika listę studentów (imię, nazwisko, grupa) i zapisze ją do pliku „lista.txt”
b. b. Jeżeli na dysku jest już plik „lista.txt”, wczyta z niego listę studentów (imię, nazwisko, grupa) , a użytkownikowi umożliwi dopisywanie nowych studentów do listy, na koniec zapisze listę z powrotem do pliku.

Ćwiczenie III. Napisz program „losuj_plik.py”, który wczyta od użytkownika trzy liczby całkowite: *a*, *b* i *n*, a następnie wygeneruje plik, zawierający *n* linii, z których w każdej znajdzie się losowa liczba całkowita z zakresu od *a* do *b*. W różnych liniach mają znaleźć się różne liczby, jednak ta sama liczba może występować w pliku więcej niż raz.
Ćwiczenie IV. Napisz program „losuj_plik2.py”, różniący się od programu z ćwiczenia III tym, że w całym pliku określona liczba może wystąpić tylko jeden raz. Plik nie może mieć przy tym więcej linii, niż wynosi długość zakresu od *a* do *b*.

LEKCJA 13 - OPERACJE NA PLIKACH I KATALOGACH

Jednym z podstawowych zadań systemu operacyjnego jest obsługa dyskowego systemu plików. Na poprzedniej lekcji omówiono funkcje Pythona służące do obsługi plików jako zbiorów danych, ich tworzenia i modyfikacji. W tym podrozdziale zostaną omówione funkcje służące do manipulacji plikami w całości, ich przenoszenia i usuwania oraz funkcje obsługujące katalogi dyskowe.
Funkcje, którymi będziemy się zajmować zawarte są w module standardowym os:

```
>>> from os import *
```

Aby sprawdzić, w jakim działamy obecnie katalogu dyskowym używamy funkcji getcwd:

```
>>> getcwd()
'C:\\Python24'
```

Aby zmienić bieżący katalog dyskowy na inny, używamy funkcji chdir(*nowy katalog*):

```
>>> chdir('tcl')
>>> getcwd()
'C:\\Python24\\tcl'
```

Aby poznać zawartość dowolnego katalogu dyskowego, używamy funkcji listdir(*katalog*); dla bieżącego katalogu:

```
>>> listdir('.')
['tcl84.lib', 'tclstub84.lib', 'tix8184.lib', 'tk84.lib', 'tkstub84.lib', 'tk8.4', 'tix8.1', 'tcl8.4', 'reg1.1', 'dde1.2']
```

Dla podkatalogu 'tcl8.4':

```
>>> listdir('tcl8.4')
['auto.tcl', 'history.tcl', 'init.tcl', 'ldAout.tcl', 'package.tcl', 'parray.tcl', 'safe.tcl', 'tclIndex', 'word.tcl', 'tcltest2.2', 'opt0.4',
'msgcat1.3', 'http2.4', 'http1.0', 'encoding']
```

Oczywiście, równie dobrze możemy podać ścieżkę absolutną katalogu:

```
>>> listdir(r'C:\Python24\tcl\tcl8.4')
['auto.tcl', 'history.tcl', 'init.tcl', 'ldAout.tcl', 'package.tcl', 'parray.tcl', 'safe.tcl', 'tclIndex', 'word.tcl', 'tcltest2.2', 'opt0.4',
'msgcat1.3', 'http2.4', 'http1.0', 'encoding']
```

Aby filtrować pliki i katalogi według określonego wzorca, musimy posłużyć się funkcją fnmatch(*nazwa*, *wzorzec*) z modułu fnmatch. Funkcja ta zwraca prawdę, wtedy i tylko wtedy, gdy *nazwa* odpowiada *wzorcowi*:

```
>>> import fnmatch
>>> fnmatch.fnmatch('Python','P*n')
True
>>> fnmatch.fnmatch('Python','P*e')
False
```

Lista plików z rozszerzeniem ‘tcl’:

```
>>> [x for x in listdir(r'C:\Python24\tcl\tcl8.4') if \
fnmatch.fnmatch(x,'*.tcl')]
['auto.tcl', 'history.tcl', 'init.tcl', 'ldAout.tcl', 'package.tcl', 'parray.tcl', 'safe.tcl', 'word.tcl']
```

Lista plików o nazwach kończących się na ‘t’ lub ‘y’:

```
>>> [x for x in listdir(r'C:\Python24\tcl\tcl8.4') if \
fnmatch.fnmatch(x,'*[ty].*')]
```

['history.tcl', 'init.tcl', 'ldAout.tcl', 'parray.tcl']

Zbliżony rezultat otrzymamy używając funkcji glob z modułu glob:

```
>>> for x in glob.glob(r'C:\Python24\tcl\tcl8.4\*[ty].*'):
        print x

C:\Python24\tcl\tcl8.4\history.tcl
C:\Python24\tcl\tcl8.4\init.tcl
C:\Python24\tcl\tcl8.4\ldAout.tcl
C:\Python24\tcl\tcl8.4\parray.tcl
```

Jak widać, ścieżki do znalezionych obiektów dyskowych zwracane są w takiej postaci, w jakiej podano jej we wzorcu (w tym przypadku absolutne). Aby rozdzielić ścieżkę absolutną na katalog zawierający plik i nazwę pliku używamy funkcji path.split:

```
>>> path.split('C:\Python24\tcl\tcl8.4\history.tcl')
('C:\\Python24\tcl\tcl8.4', 'history.tcl')
>>> for x in glob.glob(r'tcl8.4\*[ty].*'):
        print path.split(x)[1]


history.tcl
init.tcl
ldAout.tcl
parray.tcl
```

Funkcja path.join łączy ciąg katalogów w ścieżkę:

```
>>> path.join('C:','Python24','tcl','tcl8.4','history.tcl')
'C:Python24\\tcl\\tcl8.4\\history.tcl'
>>> path.join(r'C:\Python24','tcl','tcl8.4\history.tcl')
'C:\\Python24\\tcl\\tcl8.4\\history.tcl'
```

Funkcja path.isabs sprawdza, czy podana ścieżka jest absolutna:

```
>>> path.isabs(r'tcl8.4\history.tcl')
False
>>> path.isabs(r'C:\Python24\tcl\tcl8.4\history.tcl')
True
```

Funkcja path.exists sprawdza, czy dany obiekt dyskowy istnieje:

```
>>> path.exists('C:\\Python24\\tcl\\tcl8.4\\history.tcl')
True
>>> path.exists('C:\\Python24\\tcl\\nowy')
False
```

Funkcja mkdir(*nazwa katalogu*) tworzy na dysku nowy katalog:

```
>>> mkdir('nowy')
>>> path.exists('C:\\Python24\\tcl\\nowy')
True
>>> listdir('.')
['tcl84.lib', 'tclstub84.lib', 'tix8184.lib', 'tk84.lib', 'tkstub84.lib', 'tk8.4', 'tix8.1', 'tcl8.4', 'reg1.1', 'dde1.2', 'nowy']
```

Funkcja rename(*dotychczasowa nazwa*, *nowa nazwa*) zmienia nazwę pliku lub katalogu:

```
>>> rename('stary','nowy')
>>> path.exists('C:\\Python24\\tcl\\nowy')
```

```
False
>>> listdir('.')
['tcl84.lib', 'tclstub84.lib', 'tix8184.lib', 'tk84.lib', 'tkstub84.lib', 'tk8.4', 'tix8.1', 'tcl8.4', 'reg1.1', 'dde1.2', 'stary']
```

Może także służyć do przenoszenia obiektów dyskowych pomiędzy katalogami:

```
>>> file('plik','w').close()
>>> listdir('.')
['tcl84.lib', 'tclstub84.lib', 'tix8184.lib', 'tk84.lib', 'tkstub84.lib', 'tk8.4', 'tix8.1', 'tcl8.4', 'reg1.1', 'dde1.2', 'stary', 'plik']
>>> rename('plik',r'stary\plik')
>>> path.exists('C:\\Python24\\tcl\\plik')
False
>>> path.exists('C:\\Python24\\tcl\\stary\\plik')
True
>>> rename(r'stary\plik','plik')
>>> path.exists('C:\\Python24\\tcl\\stary\\plik')
False
>>> path.exists('C:\\Python24\\tcl\\plik')
True
```

Funkcja path.isfile sprawdza, czy dany obiekt dyskowy jest plikiem:

```
>>> path.isfile('stary')
False
>>> path.isfile('plik')
True
```

Funkcja path.isdir sprawdza, czy dany obiekt dyskowy jest katalogiem:

```
>>> path.isdir('plik')
False
>>> path.isdir('stary')
True
```

Funkcja path.ismount sprawdza, czy dany obiekt dyskowy jest dyskiem:

```
>>> path.ismount('stary')
False
>>> path.ismount('C:\\')
True
```

Funkcja path.getsize zwraca długość pliku w bajtach:

```
>>> path.getsize('plik')
0L
>>> f=file('plik','w'); f.write('Siedem!'); f.close()
>>> path.getsize('plik')
7L
>>> f=file('plik','w'); f.write('*'*100); f.close()
>>> path.getsize('plik')
100L
```

Dla katalogów zwracane jest zawsze zero:

```
>>> path.getsize('stary')
0L
>>> for x in listdir('.'):
        print x, path.getsize(x)

tcl84.lib 190886
tclstub84.lib 2272
```

```
tix8184.lib 49054
tk84.lib 165420
tkstub84.lib 2996
tk8.4 0
tix8.1 0
tcl8.4 0
reg1.1 0
dde1.2 0
stary 0
plik 100
```

Funkcja path.getctime zwraca czas stworzenia, a path.getmtime czas ostatniej modyfikacji obiektu dyskowego:

```
>>> from time import ctime
>>> ctime(path.getctime('plik'))
'Thu Oct 12 12:26:50 2006'
>>> ctime(path.getmtime('plik'))
'Thu Oct 12 12:33:16 2006'
```

Funkcja walk(*katalog nadrzędny, kolejność*) służy do rekursywnego przechodzenia podanego katalogu wraz ze wszystkimi jego podkatalogami. Dla każdego znalezionego katalogu (łącznie z *nadrzędnym*) zwraca trójkę: (*ścieżka*, *podkatalogi*, *pliki*), gdzie *ścieżka* oznacza ścieżkę dostępu do katalogu, *podkatalogi* – listę nazw zawartych w nim podkatalogów, *pliki* – listę nazw zawartych w nim plików. Parametr *kolejność* (domyślnie True) ustawiony na False zwraca katalogi w odwrotnym porządku (zaczyna od najgłębiej położonego; może być to przydatne, jeżeli zamierzamy np. kasować podkatalogi).

```
>>> for sciezka, podkatalogi, pliki in walk(r'C:\Python24\Tcl'):
        print 'W katalogu %s znajduje się %i bajtów w %i plikach' \
% (sciezka, sum(path.getsize(path.join(sciezka, nazwa)) \
for nazwa in pliki), len(pliki))

W katalogu C:\Python24\Tcl znajduje się 410728 bajtów w 6 plikach
W katalogu C:\Python24\Tcl\tk8.4 znajduje się 436678 bajtów w 30 plikach
W katalogu C:\Python24\Tcl\tk8.4\msgs znajduje się 51740 bajtów w 12 plikach
W katalogu C:\Python24\Tcl\tk8.4\images znajduje się 97217 bajtów w 13 plikach
W katalogu C:\Python24\Tcl\tk8.4\demos znajduje się 269352 bajtów w 54 plikach
W katalogu C:\Python24\Tcl\tk8.4\demos\images znajduje się 277824 bajtów w 11 plikach
W katalogu C:\Python24\Tcl\tix8.1 znajduje się 596476 bajtów w 77 plikach
W katalogu C:\Python24\Tcl\tix8.1\pref znajduje się 236295 bajtów w 36 plikach
W katalogu C:\Python24\Tcl\tix8.1\bitmaps znajduje się 18805 bajtów w 58 plikach
W katalogu C:\Python24\Tcl\tcl8.4 znajduje się 121278 bajtów w 9 plikach
W katalogu C:\Python24\Tcl\tcl8.4\tcltest2.2 znajduje się 98660 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\tcl8.4\opt0.4 znajduje się 33631 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\tcl8.4\msgcat1.3 znajduje się 13083 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\tcl8.4\http2.4 znajduje się 24706 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\tcl8.4\http1.0 znajduje się 10494 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\tcl8.4\encoding znajduje się 1413736 bajtów w 78 plikach
W katalogu C:\Python24\Tcl\reg1.1 znajduje się 13182 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\dde1.2 znajduje się 13646 bajtów w 2 plikach
W katalogu C:\Python24\Tcl\stary znajduje się 0 bajtów w 0 plikach
```

Funkcja remove usuwa z dysku plik, a rmdir katalog o podanej nazwie:

```
>>> remove('plik')
>>> path.exists('plik')
False
>>> rmdir('stary')
>>> path.exists('stary')
False
```

Ćwiczenie 1. Napisz program, który znajdzie w podanym przez użytkownika katalogu i wszystkich jego podkatalogach wszystkie pliki, które zawierają podany przez użytkownika napis.
Ćwiczenie 2. Napisz program, który znajdzie w podanym przez użytkownika katalogu i wszystkich jego podkatalogach najstarszy i najdłuższy plik.
Ćwiczenie 3. Napisz program, który znajdzie w podanym przez użytkownika katalogu i wszystkich jego podkatalogach wszystkie zdublowane pliki, czyli takie pliki, których nazwa występuje jednocześnie w więcej niż jednym miejscu.

LEKCJA 14 – PROSTA BAZA DANYCH

Wiemy już jak zapisywać w plikach napisy. Co jednak należałoby zrobić, gdybyśmy chcieli zapisać w pliku obiekty innych typów? Istnieje co prawda funkcja `str`, zamieniająca obiekt standardowego typu na postać tekstową:

```
lista = [1, 2, "trzy", 4]
>>> s=str(lista)
>>> s
"[1, 2, 'trzy', 4]"
```

Problem w tym, że nie da się łatwo wykonać odwrotnej transformacji, a dokładniej rzecz ujmując, jej rezultat jest daleki od pożądanego:

```
>>> l=list(s)
>>> l
['[', '1', ',', ' ', '2', ',', ' ', "'", 't', 'r', 'z', 'y', "'", ',', ' ', '4', ']']
```

Na szczęście w Pythonie dostępny jest moduł `pickle`, służący, jak nazwa wskazuje, do peklowania obiektów. Programiści piszący w językach .NET lub Javie używają na określenie tego procesu bardziej górnolotnego terminu, a mianowicie mówią o serializacji. **Serializacja obiektu** polega na przekształceniu danych go opisujących w ciąg bajtów (funkcja `dumps`), z którego można później odtworzyć taki sam obiekt (funkcja `loads`).

```
>>> import pickle
>>> zapis=pickle.dumps(lista)
>>> l=pickle.loads(zapis)
>>> l
[1, 2, 'trzy', 4]
```

Jak widać powyżej, udało nam się zachować i odtworzyć listę w zmiennej `zapis`. Sam `zapis` jest napisem o poniższej zawartości:

```
>>> zapis
"(lp0\nI1\naI2\naS'trzy'\np1\naI4\na."
```

Załóżmy teraz, że chcielibyśmy razem z listą zachować i słownik. To również nie jest trudne, pod warunkiem umieszczenia ich w krotce:

```
>>> slownik={"a":"b",1:2}
>>> zapis=pickle.dumps((lista,slownik))
>>> del lista
>>> del slownik
>>> (lista,slownik)=pickle.loads(zapis)
>>> lista; slownik
[1, 2, 'trzy', 4]
{'a': 'b', 1: 2}
```

Dzięki `pickle` możemy także zachowywać obiekty należące do klas zdefiniowanych przez nas samych:

```
>>> class wymiary3:
      x=0; y=0; z=0

>>> w3=wymiary3()
>>> w3.x=1; w3.y=2; w3.z=3
>>> zapis=pickle.dumps(w3)
>>> del w3
>>> w3=pickle.loads(zapis)
>>> w3.x; w3.y; w3.z
1
2
3
```

Napis reprezentujący zapeklowany obiekt możemy zapisać samodzielnie do pliku, możemy też posłużyć się funkcjami `dump` i `load`, które (w odróżnieniu od `dumps` i `loads`) zachowują obiekt w pliku (a nie napisie):

```
>>> f1=file("trzy_rzeczy.txt","w+")
>>> pickle.dump((lista,slownik,w3),f1)
>>> lista=[]; slownik={}; w3=wymiary3()
>>> lista; slownik; w3.x
[]
{}
0
>>> f1.seek(0)
>>> (lista,slownik,w3)=pickle.load(f1)
>>> lista; slownik; w3.x
[1, 2, 'trzy', 4]
{'a': 'b', 1: 2}
1
```

Zachowywanie wielu obiektów w pojedynczej krotce jest wygodne, dopóki ich liczba nie osiągnie zbyt dużej wartości. Wtedy o wiele wygodniejsze jest użycie słownika. Najprostszym takim rozwiązaniem dostępnym w Pythonie jest baza danych zdefiniowana w module `dumbdbm`, stanowiąca w istocie implementację pliku o organizacji indeksowo-sekwencyjnej. Metoda `dumbdbm.open` tworzy na dysku (lub otwiera istniejącą) prostą bazę danych o podanej nazwie (w istocie na dysku tworzone są dwa pliki: indeksowany z rozszerzeniem „.dat" i indeksujący z rozszerzeniem „.dir"). Obsługa bazy jest identyczna jak obsługa słownika, z tą różnicą, że wszystkie zachowane w niej dane przechowywane są nie w pamięci, lecz na dysku:

```
>>> import dumbdbm
>>> db=dumbdbm.open("prosta_baza")
>>> db['napis']="hej ho!"
>>> db['napis']
'hej ho!'
```

Bazy danych typu `dbm` pozwalają używać tylko napisów jako kluczy (co jest do przyjęcia) i wartości (co stanowi pewien problem). Stąd próba zachowania w niej obiektu innego niż napis typu, nieuchronnie kończy się błędem:

```
>>> db['lista']=lista

Traceback (most recent call last):
  File "<pyshell#282>", line 1, in -toplevel-
    db['lista']=lista
  File "C:\Python24\lib\dumbdbm.py", line 160, in __setitem__
  raise TypeError, "keys and values must be strings"
TypeError: keys and values must be strings
```

Rozwiązaniem jest oczywiście peklowanie, jednak w praktyce nie jest to zbyt wygodne:

```
>>> db['lista']=pickle.dumps(lista)
>>> pickle.loads(db['lista'])
[1, 2, 'trzy', 4]
```

dlatego naszą prostą bazę już zamkniemy

```
>>> db.close()
```

a zajmiemy się bliżej modułem `shelve`, który oferuje analogiczny sposób dostępu do danych (podobny słownikowi), umożliwiając jednak zachowywanie obiektów dowolnego typu (nie tylko napisów):

```
>>> import shelve
>>> db = shelve.open ('baza')
>>> db['lista']=lista
>>> db['lista']
```

```
[1, 2, 'trzy', 4]
>>> db['slownik']=slownik
>>> db['slownik']
{'a': 'b', 1: 2}
```

W opisywanej wersji Pythona, `shelve` posługuje się lepszym niż `dumbdbm` motorem bazy danych, a mianowicie `dbhash` (który z kolei opiera się na motorze `BSD`). Nadal jednak jest to motor nie pozwalający na obsługę dużych baz danych. W przypadku takiej konieczności właściwym rozwiązaniem jest podłączenie Pythona do zewnętrznej bazy danych (np. poprzez sterowniki ODBC), co również jest czynnością prostą, jednak z pewnością wykraczającą poza podstawy programowania (dla naszych skromnych potrzeb w zupełności wystarczający jest już `dumbdbm`), stąd problematyki tej nie będziemy tu podejmować, odsyłając zainteresowanych do specjalistycznej literatury.
Bazę danych stworzoną przy pomocy `shelve` obsługujemy tak jak słownik, a zatem dostępne są wszystkie operacje i metody działające dla prawdziwych słowników:

- Ilość elementów:

```
>>> len(db)
2
```

- Sprawdzenie klucza:

```
>>> 'lista' in db
True
```

- Lista kluczy:

```
>>> db.keys()
['lista', 'slownik']
```

- Lista wartości:

```
>>> db.values()
[[1, 2, 'trzy', 4], {'a': 'b', 1: 2}]
```

- Lista kluczy i wartości:

```
>>> db.items()
[('lista', [1, 2, 'trzy', 4]), ('slownik', {'a': 'b', 1: 2})]
```

- Modyfikacja wartości:

```
>>> db['lista']=[3,2]
```

- Usunięcie elementu:

```
>>> del db['lista']
>>> db.items()
[('slownik', {'a': 'b', 1: 2})]
```

- Usunięcie wszystkich elementów:

```
>>> db.clear()
>>> db.items()
[]
```

Ponadto, bazę można zamknąć:

```
>>> db.close()
```

Przyjrzymy się teraz programowi, w którym wykorzystano opisane wyżej rozwiązania.
Przykład. Program ‘parking.py’ służy do ewidencjonowania samochodów stojących na płatnym parkingu. Realizuje następujące funkcje: wjazd (zapisanie numeru samochodu i godziny zaparkowania), wyjazd (oblicza opłatę należną za czas parkowania), ustalenie opłaty i okresu jej naliczania, wyświetlenie listy pojazdów stojących aktualnie na parkingu.
Kod źródłowy z opisem. Na początku otwieramy w IDLE nowe okno edycji i od razu zapisujemy pod nazwą ‘parking.py’.
Program korzystał będzie z trzech modułów: `shelve` – do obsługi bazy danych, `sys` – do wychodzenia z programu, `time` – do obsługi czasu. Piszemy więc:

```
import shelve, sys
from time import *
```

Każdą z funkcji programu ujmiemy w osobnym podprogramie. Zaczniemy od zmiany wysokości opłaty parkingowej.

```
# zmiana opłat
def zmiana_stawki():
    global baza,stawka,okres   # zmienne globalne
    print "\nZmiana wysokości opłat\n"
    print "Bieżąca stawka wynosi %.2f zł za %i minut(y)\n" % (stawka,okres)
    try:
        s=float(raw_input("Podaj nową wysokość opłat: "))
        o=int(raw_input("Podaj nowy czas naliczania w minutach: "))
    except:
        print "Błąd wprowadzania danych! Stawka nie została zmieniona!"
        return
    try:
        baza['_stawka']=(s,o)  # zapisujemy w bazie
    except:
        print "Błąd zapisu danych! Stawka nie została zmieniona!"
    else:
        stawka=s    # kopiujemy do
        okres=o     # zmiennych globalnych
```

Zmienne `stawka` i `okres` są globalne, gdyż korzystać z nich będą również inne funkcje. Wprowadzone wartości próbujemy zapisać w bazie, a dopiero kiedy to się uda – kopiujemy do zmiennych globalnych (taka kolejność jest konieczna by zachować spójność danych w bazie i pamięci w każdym przypadku).
Inicjalizacja bazy danych ma za zadanie otworzyć bazę i załadować z niej wcześnie zapisaną stawkę. Jeżeli baza jest świeża, stawka musi być wprowadzona własnoręcznie przez użytkownika (poprzez wywołanie funkcji zdefiniowanej przed chwilą; wcześniej, aby w niej uniknąć błędu, inicjalizujemy zmienne globalne pierwszymi lepszymi wartościami).
Stawkę zapisujemy w kluczu `'_stawka'`. Możemy sobie na to pozwolić, przyjmując, że nie jest to dopuszczalny numer rejestracyjny w Polsce.

```
#inicjalizacja bazy danych
def init():
    global baza,stawka,okres
    try:
        baza = shelve.open ('baza_parkingowa') # otwarcie
    except:
        print "Błąd krytyczny! Baza danych nie została otwarta!"
        sys.exit(0) # wyjście z programu
    print "Inicjalizacja udana. Baza danych została otwarta."
    if '_stawka' in baza.keys():    # czy już istnieje?
        (stawka,okres)=baza['_stawka']  # tak - kopiujemy stawki
    else:
        (stawka,okres)=(0.0,1)  # nie -
        zmiana_stawki()    # wczytujemy od użytkownika
```

Kolejna funkcja zajmuje się wyświetlaniem menu głównego programu. Wyświetla ono dostępne funkcje i daje możliwość wyboru jednej z nich, zwracając rezultat na zewnątrz. Zwróćmy uwagę, że z napisu

wprowadzonego przez użytkownika bierzemy jedynie pierwszy znak i konwertujemy go na dużą literę (by uniknąć problemów, gdy użytkownik ma wyłączony klawisz CAPSLOCK).

```
# menu główne programu
def menu():
    while True:
        print
        print '-'*70
        print 'PARKING'.center(70)
        print '-'*70
        print '[W] Wjazd [E] Wyjazd [P] Pojazdy [S] Stawka [K] Koniec'.center(70)
        print '-'*70
        w=raw_input()[0].upper() # pierwszy znak (duza litera)
        if w in 'WEPSK':         # znany?
            return w             # tak - zwracamy go
        print 'Nieznane polecenie -',
```

Funkcja pojazdy wyświetla listę pojazdów znajdujących się na parkingu. Dane pobierane są z bazy i wyświetlane z użyciem prostego formatowania.

```
# lista pojazdów na parkingu
def pojazdy():
    global baza
    print
    print 'Lista pojazdów na parkingu'.center(33)
    print '-'*33
    print '|'+'Nr rej.'.center(10)+'|'+'Godz. parkowania'.center(20)+'|'
    print '-'*33
    for rej,godz in baza.items():
        if rej!='_stawka':
            print "|%9s |" % rej,strftime("%H:%M (%Y-%m-%d)",godz),'|'
    print '-'*33
```

Wyjazd pojazdu wymaga upewnienia się czy dany pojazd rzeczywiście był zaparkowany, następnie obliczenia należnej opłaty, a wreszcie usunięcia pojazdu z bazy. Aby wyliczyć opłatę musimy: zamienić czas parkowania i obecny na sekundy (przy użyciu `mktime`), wyliczyć ich różnicę, przeliczyć na minuty (60 sekund w minucie), przeliczyć na jednostki taryfowe (uwzględniając zasadę zaliczania każdej rozpoczętej jednostki – dodajemy po prostu liczbę minut o 1 mniejszą od pełnego okresu), a na końcu przemnożyć rezultat przez wysokość opłaty.

```
# rejestracja wyjazdu pojazdu
def wyjazd():
    global baza,stawka,okres
    print 'Wyjazd pojazdu - godzina',strftime("%H:%M (%Y-%m-%d)")
    rej=raw_input('Podaj numer rejestracyjny pojazdu: ')
    if rej in baza.keys():    # czy taki był zaparkowany?
        godz=baza[rej]
    print "Godzina wjazdu:",strftime("%H:%M (%Y-%m-%d)",godz)
        minuty=int(mktime(localtime())-mktime(godz))/60
        jednostki=(minuty+okres-1)/okres # naliczamy za rozpoczeta
        print "\nDo zapłaty: %.2f zł" % (jednostki*stawka)
        del baza[rej] # usuwamy wpis
    else:
        print "Błąd! Takiego pojazdu nie ma na parkingu!"
```

Rejestracja wjazdu pojazdu jest znacznie prostsza, wymaga jedynie upewnienia się, czy dany pojazd aby nie był już zaparkowany i dopisania numeru rejestracyjnego oraz aktualnego czasu do bazy. Przyjmujemy numery rejestracyjne nie dłuższe niż 9 znaków; nie przyjmujemy rejestracji `'_stawka'`, by uniemożliwić uszkodzenie zapisu stawek.

```
# rejestracja wjazdu pojazdu
def wjazd():
    global baza
```

```
    godz=localtime()
    print 'Wjazd pojazdu - godzina',strftime("%H:%M (%Y-%m-%d)",godz)
    rej=raw_input('Podaj numer rejestracyjny pojazdu: ')[:9]
    if rej=='_stawka': return      # zabezpieczenie
    if rej not in baza.keys():    # nie jest zaparkowany?
        baza[rej]=godz
        print "Wprowadzono."
    else:
        print "Błąd! Taki pojazd już jest na parkingu!"
```

Z programu głównego wyłączyliśmy jeszcze funkcję wywołującą odpowiedni podprogram w zależności od wyboru użytkownika.

```
# realizacja wyboru użytkownika
def wybor():
    while True:
        w=menu()
        if w=='K':
            break
        elif w=='S':
            zmiana_stawki()
        elif w=='P':
            pojazdy()
        elif w=='E':
            wyjazd()
        elif w=='W':
            wjazd()
```

Sam program główny jedynie otwiera bazę, wywołuje funkcję `wybor`, a na końcu zamyka bazę.

```
# program główny
init()          # otwarcie bazy
try:
    wybor()         # interfejs użytkownika
except:
    print "Wystąpił poważny błąd."
baza.close()    # zamknięcie bazy
```

Przykład uruchomienia. Wciskamy przycisk F5. Jako, że jest to pierwsze uruchomienie programu, zostaniemy poproszeni o podanie wysokości opłat.

```
>>> ================================ RESTART ================================
>>>
Inicjalizacja udana. Baza danych została otwarta.

Zmiana wysokości opłat

Bieżąca stawka wynosi 0.00 zł za 1 minut(y)

Podaj nową wysokość opłat:
```

Wprowadzamy np. poniższe wartości (pamiętajmy o kropce dziesiętnej!).

```
Podaj nową wysokość opłat: 1.50
Podaj nowy czas naliczania w minutach: 30

------------------------------------------------------------------
                              PARKING
------------------------------------------------------------------
        [W] Wjazd [E] Wyjazd [P] Pojazdy [S] Stawka [K] Koniec
------------------------------------------------------------------
```

Kiedy pojawi się menu główne wybieramy opcję ‘W’ (potwierdzamy klawiszem ENTER) i wprowadzamy (oczywiście, godziny odpowiadają wprowadzaniu danych przez autora):

```
Wjazd pojazdu - godzina 21:10 (2006-09-18)
Podaj numer rejestracyjny pojazdu: ZS39542
Wprowadzono.
```

Kiedy ponownie pojawi się menu główne wybieramy znowu opcję ‘W’ (potwierdzamy klawiszem ENTER) i wprowadzamy drugie auto:

```
Wjazd pojazdu - godzina 21:11 (2006-09-18)
Podaj numer rejestracyjny pojazdu: SZF8510
Wprowadzono.
```

Zaparkujemy jeszcze trzecie auto:

```
Wjazd pojazdu - godzina 21:11 (2006-09-18)
Podaj numer rejestracyjny pojazdu: Z0 JOLA
Wprowadzono.
--------------------------------------------------------------------
                               PARKING
--------------------------------------------------------------------
        [W] Wjazd [E] Wyjazd [P] Pojazdy [S] Stawka [K] Koniec
--------------------------------------------------------------------
```

Tym razem z menu wybieramy opcję ‘P’. Zobaczymy (przypominam, że czasy będą się różnić) mniej więcej taki widok:

```
    Lista pojazdów na parkingu
---------------------------------
| Nr rej.  |  Godz. parkowania  |
---------------------------------
|  ZS39542 | 21:10 (2006-09-18) |
|  Z0 JOLA | 21:11 (2006-09-18) |
|  SZF8510 | 21:11 (2006-09-18) |
---------------------------------
```

Sprawdzimy teraz, czy dane rzeczywiście są przechowywane trwale. Wyjdźmy z programu wybierając z menu opcję ‘K’. Następnie uruchommy go ponownie. Powinniśmy zobaczyć:

```
>>> ================================ RESTART ================================
>>>
Inicjalizacja udana. Baza danych została otwarta.

--------------------------------------------------------------------
                               PARKING
--------------------------------------------------------------------
        [W] Wjazd [E] Wyjazd [P] Pojazdy [S] Stawka [K] Koniec
--------------------------------------------------------------------
```

Jak widać stawka została przeczytana z bazy i nie ma potrzeby jej ponownego wprowadzania. Jeżeli wybierzemy opcję ‘P’, powinniśmy zobaczyć listę identyczną jak przed wyjściem z programu:

```
    Lista pojazdów na parkingu
---------------------------------
| Nr rej.  |  Godz. parkowania  |
---------------------------------
|  ZS39542 | 21:10 (2006-09-18) |
|  Z0 JOLA | 21:11 (2006-09-18) |
|  SZF8510 | 21:11 (2006-09-18) |
---------------------------------
```

Spróbujemy teraz ‘wyjechać’ jednym z aut. Wybieramy ‘E’:

```
Wyjazd pojazdu - godzina 21:15 (2006-09-18)
Podaj numer rejestracyjny pojazdu: ZS39542
Godzina wjazdu: 21:10 (2006-09-18)
Do zapłaty: 1.50 zł
```

Sprawdzamy stan bieżący:

```
    Lista pojazdów na parkingu
---------------------------------
| Nr rej.  |  Godz. parkowania  |
---------------------------------
|  Z0 JOLA | 21:11 (2006-09-18) |
|  SZF8510 | 21:11 (2006-09-18) |
---------------------------------
```

Dalszą zabawę z programem pozostawiam Wam.

Ćwiczenia kontrolne

Ćwiczenie I. Napisz program ‘parking2.py’ różniący się od programu ‘parking.py’ tym, że klienci parkingu mogą posiadać miesięczne karty abonentowi (dodać funkcję sprzedaży). Wjazdy i wyjazdy samochodów takich klientów są rejestrowane, jednak przy wyjeździe z parkingu opłata nie jest pobierana, o ile nie abonament nie skończył się (wyświetla się tylko informacja o liczbie pozostałych dni). Jeżeli abonament danego auta skończył się, przy jego najbliższym wjeździe lub wyjeździe użytkownik jest o tym informowany i może wykupić nowy abonament lub z niego zrezygnować (i w konsekwencji wnieść standardową opłatę).

Ćwiczenie II. Napisz program ‘narzedzia.py’ służący do obsługi narzędziowni. Program ma umożliwiać wykonywanie następujących funkcji: dopisanie nowego narzędzia na listę, zmiana liczby sztuk narzędzia na liście (w tym możliwość kompletnego usunięcia narzędzia, gdy liczba sztuk spadnie do zera), wydanie narzędzia (zapamiętanie godziny i nazwiska pracownika biorącego narzędzie), zwrot narzędzia, lista wszystkich narzędzi, lista dostępnych narzędzi, lista wydanych narzędzi.