LWP: Lamerskie Wprowadzenie do Pythona

Artur Skura <arturs@linuxpl.org>

podziękowania dla Wojtka Warczakowskiego <wowar@poczta.onet.pl>

gdzieniegdzie tekst modyfikował Maciej " MACiAS" Pilichowski <macias@torun.pdi.net>

07.07.2001

szablon DocBook; LyX 1.1.6fix2

Dokument ten przedstawia podstawowe właściwości potężnego języka, jakim jest Python. Przeznaczony jest dla nie−programistów i hobbystów. Proszę zgłaszać wszystkie błędy (zapewne jest ich mnóstwo) i nie irytować się łopatologią ;−)

---

1. Wstęp

Wiele osób pragnąc nauczyć się programować staje przed dylematem: " Od jakiego języka programowania zacząć?" . Pytanie jest dość istotne, ponieważ negatywne nawyki, którymi przesiąknie początkujący bardzo trudno potem wykorzenić.

Eric S. Raymond w swoim eseju " Jak zostać hackerem?" proponuje Pythona. Istnieje wiele argumentów przemawiających za wykorzystaniem tego języka. Niektóre aspekty C (np. wskaźniki) mogą sprawić początkującym wiele trudności; utrzymanie porządku w programach napisanych w Perlu staje się po pewnym czasie bardzo trudne (podobnie jest z Tcl/Tk); Java wymaga zrozumienia pewnych podstaw, a narzędzia do niej −− wymagań sprzętowych wyższych niż przeciętne... Nauka Pythona jest rozsądnym kompromisem: z jednej strony otrzymujemy język, którego stosunkowo łatwo można się nauczyć, z drugiej zaś −− potężne narzędzie używane przez profesjonalistów, w swej podstawowej postaci bardzo użyteczne wszędzie tam, gdzie np. zachodzi konieczność wykonywania skomplikowanych operacji na łańcuchach znaków.

Twórca Pythona, Guido van Rossum, zaleca jego naukę na samym początku, przed wprowadzeniem do takich języków jak Java czy C++. Dzięki temu, po zakończeniu kursu uczeń będzie potrafił programować na przyzwoitym poziomie i rozumiał podstawy programowania obiektowego, co znacznie ułatwi mu naukę w/w języków.

2. Zaczynamy

Aby zorientować się w możliwościach języka, uruchomimy interpreter Pythona:

```
$ python
```

Pojawi się napis podobny do poniższego:

```
Python 1.5.2 (#1, Feb  1 2000, 16:32:16)  [GCC egcs−2.91.66 19990314/Linux (egcs− on
Copyright 1991−1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Możemy teraz wydawać polecenia, które zostaną wykonane od razu. Napiszmy:

```
>>> print "Mamusiu, jak tu pięknie!"
```

W rezultacie powinniśmy otrzymać napis " Mamusiu, jak tu pięknie!" . Python zazwyczaj robi dokładnie to, co mu każemy.

2.1 Wstępnych uwag parę

Zaczynamy gubić przykłady

Chociaż w dalszym tekście nadal jest zachowana konwencja wpisywania poleceń bezpośrednio interpreterowi Pythona, to jednak aby oszczędzić sobie czasu i nerwów możemy korzystając w edytorze tekstu (który pozwala na zapis plików ASCII) stworzyć plik o rozszerzeniu " py"

−−− np. " tescik.py" . W nim możemy pisać program w Pythonie, a po zapisaniu wydać polecenie:

```
python tescik.py
```

Jeśli popełniliśmy tzw. głupi błąd −−− literówkę −−− to wracamy do edycji pliku i poprawiamy co trzeba

Jak łatwo zauważyć...

```
print x[zk]*f/u[ol],we[d[m]]*r
```

Ok, to i tak nie jest jeszcze największy galimatias jaki można uczynić z kodu. Python pozwala Ci opisać takie kawałki własnymi słowami −−− aby odróżnić Twój opis od kodu programu użyj znaku " #" i po nim wpisz komentarz. Np:

```
# o rrrany, w ogóle nie wiem co się tutaj dzieje
```

Ale tak naprawdę powinieneś pisać kod tak, aby był samodokumentujący się −−− tj. aby faktycznie było " łatwo zauważyć, że..." nawet bez dodatkowego komentarza z Twojej strony.

A bo mnie się shift nacisnął...

Jeszcze nie wiemy, jak Python działa, ale już na wstępie:

```
>>> Q = 100
>>> print q
```

---

dostajemy piękny błąd. Dla Pythona stanowi różnicę pisownia małymi i wielkimi literami!

3. Wbudowane typy danych

3.1 Łańcuchy

Zaimplementowano w nim wiele cech innych języków eliminując związane z nimi niedogodności, dzięki czemu programista może skupić się na pracy a nie na semantyce.

Przypiszmy teraz zmiennej " a" napis " Dajcie mi kredki" :

```
>>> a = "Dajcie mi kredki."
>>> print a
Dajcie mi kredki.
```

Idźmy dalej: łączenie napisów:

```
>>> a = "Dajcie"
>>> b = "mi w końcu te kredki!!!"
>>> print a, b
Dajcie mi w końcu te kredki!!!
```

Można też inaczej:

```
>>> a = "No, "
>>> b = "wreszcie!"
>>> print a+b
No, wreszcie!
```

Można łączyć jeszcze bardziej:

```
>>> print "Jarek powiedział:", a + b
Jarek powiedział: No, Wreszcie!
```

Co ciekawe, łańcuchy można nie tylko dodawać −− można je również " mnożyć" :

```
>>> s = "Nie!"
>>> print 3*s
Nie!Nie!Nie!
```

Możemy także w prosty sposób zamienić tekst na liczbę (o ile ma to sens):

```
>>> x = "7"
>>> print x*3
777
>>> x = eval(x)    # obliczenie wartości
>>> print x*3
21
```

Gdybyśmy nie pracowali w trybie interaktywnym, a zapisywali kod do pliku, moglibyśmy

przekształcenie zapisać prościej:

```
x = ‘x‘    # odwrotne apostrofy, to samo co "
eval"
```

Ciekawym elementem Pythona są tzw. plasterki (ang. slices). Łańcuch znaków zostaje pocięty na " plasterki" , oznaczane nawiasami kwadratowymi. I tak łańcuch [0] oznacza pierwszy znak łańcucha, łańcuch [1] −− drugi, itd.:

```
>>> s = "lajkonik"
>>> print s[0]
l
>>> print s[1]
a
```

Prawda, że proste? Python idzie nieco dalej −− ogólniejsze " krojenie"

polega na wskazaniu dolnego indeksu i górnego (w ten sposób dostaniemy się do fragmentu od wskazanego dolnego indeksu do górnego, **ale bez niego**). Jeśli opuścimy, któryś z indeksów, to Python przyjmie domyślnie skrajne wartości (odpowiednio −− 0 i długość łańcucha).

Łańcuch [:2] odnosi się do pierwszych dwóch znaków łańcucha (czyli znaków o indeksie 0 i 1), zaś łańcuch [2:] −− do wszystkiego oprócz pierwszych dwóch znaków:

```
>>> print s[2:]
jkonik
>>> print s[:2]
la
```

Jak można się domyślić, łańcuch [1:5] zwraca od drugiego do piątego (od znaku o indeksie 1 do znaku o indeksie 4) znaku łańcucha:

```
>>> print s[1:5]
ajko
```

Nierzadko zdarza się, że końcowa pozycja interesuje nas w odniesieniu do długości podawanego

łańcucha, np.:

```
>>> print s[:len(s)-1]
lajkoni
```

Czyli dostaliśmy napis bez ostatniego znaku. Ale Python upraszcza nam życie jeszcze bardziej --- indeksować można z pominięciem obliczenia długości, podając samą (ujemną) wartość. Czyli:

```
>>> print s[:-1]
lajkoni
```

Czy to jasne? Poprosiliśmy Pythona o podanie łańcucha począwszy od znaku o indeksie 0 (pierwszego) do przedostatniego.

Pisanie pod lupą

Powyżej używaliśmy polecenia print z rozmaitymi operatorami. Teraz opiszemy je nieco dokładniej.

```
print "ala ma kota"
```

wypisze podany tekst i wstawi znak [enter]. Jeżeli chcemy tego uniknąć musimy postawić na samym końcu podanego tekstu znak przecinka:

```
print "ala ma kota",
```

A teraz spróbujmy w ten sposób:

```
print "ala", "ma","kota"
```

Jak powinieneś zauważyć tekst zostanie wydrukowany ze spacjami pomiędzy wyrazami. Przecinek bowiem między podanymi napisami wstawia odstęp. A jeśli chcielibyśmy tego uniknąć? Voila:

```
print "ala"+"ma"+"kota"
```

dostaniemy zbitkę słów bez rozdzielającej spacji.

3.2 Liczby

Operacje na liczbach również odbywają się w sposób intuicyjny, Rozważmy prosty przykład:

```
>>> a = 10
>>> b = 10
>>> print a, b
10 10
```

Dwom zmiennym, a i b, przypisaliśmy wartość dziesięć. Nie ma problemu ze standardowymi operacjami na liczbach:

```
>>> print a+b, a-b, a*b, a/b
20 0 100 1
```

Zaokrąglanie:

```
>>> round(10.5)
11.0
```

Potęgowanie:

```
>>> pow(5,2)
25
```

Albo i tak:

```
>>> 5 ** 2
25
```

Operacje bitowe (odpowiednio −−− bitowe or, dopełnienie, przesunięcie w lewo):

```
>>> 4|1, ~4, 4>>1
(5, −5, 2)
```

Jeśli chcemy natomiast dokonać operacji logicznych, to musimy pisać pełnymi słowami (poniższy przykład nie ma znaczenia w sensie zdrowego rozsądku):

```
>>> 4 or 2, 5 and 3
(4, 3)
```

Konwersja między różnymi systemami:

```
>>> hex(10), oct(10)
('0xa', '012')
```

Od wersji Pythona w okolicach 2.0 można stosować nienadmiarowe dodawanie (znane dobrze z C):

```
>>> x = 10
>>> x += 5
>>> print x
15
```

Zapis:

```
X operator= ...
```

jest skrótem od

```
X = X operator (...)
```

ale nie jest w 100% mu tożsamy (o tym później). W każdym razie mamy do dyspozycji wszystkie sensowne skrótowce istniejące w Pythonie także w wersji pełnej (" +=" , " −" , " *=" , etc. −−− chyba nie ma potrzeby wymieniać tu całej menażerii?).

3.3 Listy

Listy przypominają nieco tablice w innych językach. Charakteryzują się umieszczeniem elementów w nawiasach kwadratowych. Rozważmy prosty przykład:

```
>>> lista = ["Polak", "Rosjanin", "Niemiec"]
>>> print lista
['Polak', 'Rosjanin', 'Niemiec']
```

Na listach możemy dokonywać standardowych operacji: dodawać nowe elementy, usuwać stare, sortować, wyłapywać fragmenty (tak jak dla napisów −− napis jest tak naprawdę listą znaków)... Jeśli wydamy polecenie dir(nazwa typu), otrzymamy listing nazw operacji, które można przeprowadzać na danym typie:

```
>>> dir(lista)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> lista.append("Maciej")
>>> print lista
['Polak', 'Rosjanin', 'Niemiec', 'Maciej']
>>> lista.sort()
>>> print lista
['Maciej', 'Niemiec', 'Polak', 'Rosjanin']
```

Co ciekawe, w Pythonie mamy dostęp do prostej " dokumentacji"

używanych funkcji przy pomocy atrybutu .__doc__. Np.:

```
>>> print lista.append.__doc__
L.append(object) -- append object to end
```

...co się przydaje, jeśli nie jesteśmy pewni, jak użyć danej funkcji:

```
>>> print lista
['Tytus', 'Romek', 'Atomek']
>>> lista.insert.__doc__
'L.insert(index, object) -- insert object before index'
>>> lista.insert(1, "Szarik")
>>> print lista
['Tytus', 'Szarik', 'Romek', 'Atomek']
```

Jak widać używanie list jest o niebo prostsze niż w innych językach.

3.4 Zbiory (tuple) i słowniki

Na marginesie: bardzo formalnie rzecz biorąc tłumaczenie słowa " tuple" jako " zbiór" nie jest poprawne; w polskojęzycznej literaturze stosuje się inne, bardziej poprawne merytorycznie zwroty, ale są one tak szkaradne, iż nie ośmielamy się ich przytoczyć nawet małą czcionką.

Zbiór to typ danych bardzo przypominający listę, wizualnie różni się brakiem nawiasów.

```
>>> kolejka = 'Jacek', 'Wacek', 'Pankracek'
>>> print kolejka
('Jacek', 'Wacek', 'Pankracek')
>>> print kolejka[1]
Wacek
```

W przeciwieństwie do list, wartości zbiorów są niezmienne:

```
>>> kolejka[1] = 'Agatka'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Słowniki zaś przypominają tablice asocjacyjne: z każdą wartością jest skojarzony jakiś klucz. Kolejność w słowniku nie ma żadnego znaczenia, dlatego zwykłe indeksowanie nie zadziała. Najlepiej

wyjaśnić to na przykładzie:

```
>>> slownik = { 'Ania':96, 'Gosia':69}
>>> slownik['Ania']
96
```

Zobaczmy, jakich funkcji możemy użyć, by manipulować słownikami:

```
>>> dir(slownik)
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'update', 'values']
```

Keys() służy do wyświetlania kluczy:

```
>>> print slownik.keys()
['Ania', 'Gosia']
```

Zaś values() do wyświetlania wartości:

```
>>> print slownik.values()
[96, 69]
```

Do poszczególnych par możemy się dostać w ten sposób:

```
>>> print slownik.items()[1]
('Gosia', 69)
```

---

4. Podstawy sterowania programem

...odbywa się przy pomocy standardowych mechanizmów znanych z innych języków. Należy zauważyć, że w Pythonie indentacja (" wcięcia") spełniają rolę ograniczników ({}, BEGIN..END) z innych języków.

4.1 Konstrukcje warunkowe

if (jeśli)

```
>>> if 2+2==4:
...     print "2+2=4!"
...
2+2=4!
```

Należy zwrócić uwagę na dwukropek kończący warunek oraz wcięcie poprzedzające instrukcję.

elif ("jeśli natomiast...")

```
>>> if 2+2==5:
...     print "tego raczej nie wydrukujemy"
... elif 2+2==4:
...     print "a to właśnie powinniśmy zobaczyć"
```

...

Konstrukcje " if−elif−elif−..." możemy ciągnąć dość długo −− w Pythonie zastępują one znane z innych języków " case/switch" .

else ("a jeśli nie, to...")

```
>>> if 2+2==4:
...     print "Murphy powiedziałby..."
... else:
...     print "...że ten napis zostanie wyswietlony"
```

...

Murphy powiedziałby...

Ponownie warto zwrócić uwagę na wcięcia.

4.2 Konstrukcje iteracyjne

for ("dla każdego x z przedziału...")

For jest zaimplementowana w Pythonie w nieco specyficzny sposób:

```
>>> lista = [’Tytus’, ’Romek’, ’Atomek’]
>>> for i in lista:
...     print i
...
Tytus
Romek
Atomek
```

Oznacza to ni mniej nie więcej: dla każdego i, przyjmującego kolejne wartości z listy lista, wykonaj instrukcję print. Natomiast standardowe przedziały tworzy się za pomocą funkcji range():

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

A cała konstrukcja wygląda następująco:

```
>>> for i in range (1,4):
...     print "Jan Sobieski miał trzy pieski:", i
...
Jan Sobieski miał trzy pieski: 1
Jan Sobieski miał trzy pieski: 2
Jan Sobieski miał trzy pieski: 3
```

---

while ("dopóki")

```
>>> while a == 1:
...     print "OK"
```

---

rezultat powyższego zależy od wartości a. Jeśli wcześniej ustawimy a=1, pętla będzie się ciągnąć w nieskończoność. Konstrukcja while może posiadać dość specyficznie określoną akcję kończącą:

```
a = 1
while a < 10:
  a += 1 ; print a
else:
  print "koniec"
```

" Else" wykona się niezależnie od tego, czy " while" było wykonane choć jeden raz. Jaka jest w takim razie różnica dla poniższego przypadku?

```
a = 1
while a < 10:
  a += 1 ; print a
print "koniec"
```

Niewielka w zasadzie −−− inaczej jednak będzie wykonywane polecenie " break" , o którym za chwilę.

Konstrukcje for i while określa się żargonowo pętlami −−− mamy pętlę for i pętlę while.

break

Polecenie to przydaje się, kiedy chcemy natychmiast przerwać wykonywanie pętli, np.:

```
for i in range(100):
  if i==50:
    break
  print i
```

Przykład kompletnie odrealniony −−− niemniej jednak Python po dotarciu do i==50 zakończy wykonywanie pętli while. Polecenie " break"

dotyczy tej pętli, wewnątrz najbardziej której zostało umieszczone:

```
for i in range(100):
  while i<k:
    wynik = wynik + (i*k)
    k = k − 1
  if i==50:
    break
  print wynik
```

W przykładzie powyżej " break" odnosi się do pętli " for" , a nie " while" , ponieważ " while" nie zawiera " break" , natomiast for już tak. I drugi:

```
for i in range(100):
  while i<k:
    wynik = wynik + (i*k)
    if i==50:
      break
    k = k −1
  print wynik
```

Teraz z kolei " while" jest najbliższą pętlą w stosunku do " break"

(w sensie obejmowania kontroli, a nie odległości w wierszach kodu) i to tę pętlę przerwie break.

Należy podkreślić, iż konstrukcja " while−else" stanowi całość i polecenia " break" opuszcza całą konstrukcję −−− " else" nie zostanie wykonane.

continue

Działanie analogiczne do break, tyle że pętla nie jest przerywana, a jedynie pomijany jest kod po " continue" i pętla dalej kontynuuje działanie.

pass

Czyli " nic nie rób" . Nieraz przydatne.

4.3 Tajniki kontroli programu

Porównania

Pojedynczy znak równości " =" odpowiada za operację przypisania −− danej zmiennej przypisujemy jakąś wartość (niech X równa się Y). Natomiast dwa znaki równości " ==" to porównanie (czy X równa się Y?). Co innymi porównaniami −− co z nierównością? Oczywiście istnieje −− zapisujemy ją jako " <>" (tak jak w Pascalu) bądź jako " !="

(jak w C). Mamy też do dyspozycji operatory nierówności: " <" , " >" , " <=" , " >=" , a co więcej −− możemy je łączyć, np.:

```
if 10<=x<=100:
    ...
```

Kiedy wiadomo, że to prawda?

Python dokonuje skróconego sprawdzania warunków −−− jeśli w pewnym momencie okaże się, że wynik jest już ustalony, sprawdzanie jest przerywane. Np:

```
>>> x = 4
>>> y = 5
>>> if (x == 4) or (y == 777):
...     print "a jednak"
```

Napis zostanie oczywiście wyświetlony, ale nie w tym rzecz. Zgodnie z sensem operatora " or" tylko jedna z wartości musi być prawdziwa, aby wynik także był prawdziwy. W tym przypadku " x" faktycznie jest równe 4, więc jest kompletnie nieważne co następuje po " or" . Python w tym miejscu przerywa badanie i wyświetla napis.

4.4 Ułożenie kodu programu

Wcięcia

Tak jak już było to powiedziane, wcięcia mówią o " hierarchii"

danego kodu −−− wcięć możesz dokonywać przy pomocy spacji bądź tabulatora. Ilu spacji (tabulatorów)? Nieważne −−− obyś tylko zachowywał konsekwentnie swoją własną jednostkę wcięć (piszący te słowa używa wielokrotności dwu spacji, bo uważa, iż tak ładniej, a poza tym przyzwyczaił się).

Czy tekst programu nie jest przez to bardziej " goły" ? Być może −−− ale realizuje w pełni zasadę WYSIWYG. I dużo trudniej programiście pomylić się w interpretacji zagnieżdżeń w porównaniu do innych języków.

Jednolinijkowce

Do tej pory widziałeś tak zapisany program:

```
x = 4
y = 10
z = 13
```

Można jednak zapisać je łącznie, przedzielając średnikiem:

```
x = 4 ; y = 10 ; z = 13
```

Swoją drogą, gdybyś miał zwinąć taki kod:

```
x = 20
y = 20
```

to możesz zapisać to w ten sposób:

```
x = y = 20
```

ponieważ operator przypisania jest przechodni.

Co więcej proste polecenia można także pisać po instrukcjach warunkowych i pętlach, np.:

```
for i in range(100): print i
```

5. Funkcje

5.1 Podstawy

Tworzenie funkcji w Pythonie jest bardzo proste. Zaczynają się od słowa def, po którym następuje nazwa funkcji z nawiasami, w których opcjonalnie mogą się znajdować argumenty.

```
>>> def dodaj(a,b):
...     return a+b
...
>>> dodaj(1,2)
3
```

Nie musimy martwić się o sposób przekazywania argumentów (?):

```
>>> def zamien(a,b):
...     temp=a
...     a=b
...     b=temp
...     return a,b
...
>>> zamien(10,11)
(11, 10)
```

5.2 Argumenty

Argumenty domyślne

Funkcjom możemy przypisać domyślne argumenty:

```
>>> def pytanie(odpowiedz="Tito"):
...     return odpowiedz
...
```

Jeśli teraz wywołamy funkcję z argumentem, zostanie on uwzględniony:

```
>>> pytanie('Lenin')
'Lenin'
```

...a jeśli nie, przyjęta zostanie wartość domyślna:

```
>>> pytanie()
'Tito'
```

Argumenty nazwane

Python pozwala na bardzo eleganckie przypisywanie wartości pewnym argumentom. Powiedzmy, że

mamy taką oto funkcję:

```
>>> def okno(x=0,y=0,szerokosc=400,wysokosc=200,ramka=1,kolor=kGranatowy,czcionka=Ti
```

I chcemy tę funkcję wywołać, ale w zasadzie wystarczają nam wartości domyślne za wyjątkiem czcionki. Oto jak możemy postąpić przy wywołaniu:

```
>>> okno(czcionka=Arial)
```

Na marginesie −−− jeśli programowaliście kiedyś na Amidze, może dostrzeżecie pewne podobieństwo d argumentów w postaci tag−list.

5.3 Dane proste i wskazywane

Popatrzmy na poniższy przykład:

```
>>> def wypelnij_liste(lista):
...     lista = ["czyzby","kotek","zginal"]
...
>>> lista = ["ala","ma","kotka"]
>>> wypelnij_liste(lista)
>>> print lista
```

Jak sądzisz, co zobaczysz na ekranie? To dziwne na pozór zachowanie łatwo zrozumieć, jeśli przyjmiesz, że listy, słowniki i zbiory nie zawierają danych −−− one tylko na nie **wskazują**. Polecenie

```
lista = ["ala","ma","kotka"]
```

nie powoduje przechowania tych trzech napisów **w** zmiennej lista. Napisy są przechowane w pamięci komputera, a " lista" na nie wskazuje (efekt użycia operatora przyrównania −− " ="). Jeśli nie widzisz jeszcze różnicy, to pomyśl o tabliczce z napisem " Toruń 100km" −−− taka tabliczka nie zawiera w sobie całego miasta, ona tylko nań wskazuje!

Wywołanie funkcji nie zmienia tego obszaru pamięci (on nadal istnieje), tworzony jest jedynie nowy

obszar pamięci z innymi danymi, zmienna " lista" na chwilę wskazuje na nowy obszar, po czym wracając z funkcji przywraca oryginalne wskazanie.

Oczywiście z tego samego powodu w poniższym przykładzie zmieni się nam właściciel futrzaka:

```
>>> def wypelnij_liste(lista):
...     lista[0] = "tomek"
...
>>> lista = ["ala","ma","kotka"]
>>> wypelnij_liste(lista)
>>> print lista
```

Tym razem nie ingerowaliśmy we wskazanie do danych (które miałoby i tak efekt lokalny), ale dokładnie w obszar pamięci. No dobrze, ale jak zabezpieczyć się przed zmianami na liście? To proste −− skopiować obszar danych, np. tak jak poniżej:

```
>>> def wypelnij_liste(lista_arg):
...     lista = lista_arg[:]    # tutaj tworzymy <bf>kopię argumentu
...     lista[0] = "tomek"
...
>>> lista = ["ala","ma","kotka"]
>>> wypelnij_liste(lista)
>>> print lista
```

Tym razem kotek nie zmienił właściciela.

Jeśli programujesz np. w C, to może już widzisz pełną analogię między traktowaniem argumentów struktur w Pythonie, a przekazywaniem w C argumentu jako wskaźnika do danych (np. klasyczne równouprawnienie tablicy i wskaźnika do niej; w pewnym sensie rzecz jasna). W ogólności w Pythonie istnieją argumenty niemodyfikowalne (czyli odpowiadające w C przekazywaniu przez wartość) oraz modyfikowalne, zmienne (odpowiadające w C przekazywaniu przez wskaźnik).

5.4 Zasięg nazw

Python w doborze nazw zmiennych kieruje się zasadą −−− " bliższa koszula ciału" . Oznacza, to że jeśli jest to możliwe, Python będzie interpretował zmienną lokalnie, jeśli nie −−− globalnie, a i jeśli tu nie −−· jak wbudowaną. Pamiętając o tym, iż Python dynamicznie tworzy zmienne, przyjrzyjmy się następującym wariantom kodu:

```
tytul = "ala"
```

```
def zmien():
  tytul = "zuzia"    # ten wiersz będziemy omawiać
  print "funkcja", tytul

zmien()
print "program", tytul
```

W oznaczonym przypisaniu będzie użyta zmienna lokalna. Polecenie to może wykreować zmienną lokalną, więc w zgodzie z zasadą wyboru zasięgu, Python ją wykreuje. Jeśli chcielibyśmy odwołać się tu do zmiennej globalnej powinniśmy zacząć funkcję tak:

```
def zmien():
  global tytul
```

Polecenie " global" zmienia interpretację dla wskazanej zmiennej. Nie musimy stosować polecenia " global" , jeśli użycie zmiennej wymusza taką właśnie interpretację:

```
def zmien():
  podtytul = tytul + " ma kota"
  print "funkcja", podtytul
```

W tym przypadku nie ma cienia wątpliwości, iż chodzi o zmienną globalną " tytul" −−− lokalna interpretacja jest niepoprawna, gdyż brak jest przypisania (do zmiennej " tytul"). A co w takim przypadku:

```
def zmien():
  tytul = tytul + " ma kota"
  print "funkcja", tytul
```

Dostaniemy błąd −−− lewa część przypisania wskazuje na użycie lokalne, ale prawa na globalne. Przeważy interpretacja lokalna (dla całego wiersza) i przy próbie wykonania program oburzy się, iż próbowaliśmy użyć nieokreślonej zmiennej (rzecz dotyczy prawej części przypisania).

6. Moduły

Moduły zawierają wiele dodatkowych funkcji. Wraz z Pythonem dostarczanych jest wiele modułów, oferujących ogromne choć rzadko doceniane możliwości. Ładuje się je przy pomocy polecenia import.

6.1 string −− łańcuchy znaków

Moduł string dostarcza wielu funkcji do manipulacji łańcuchami znaków. Zaimportujmy go:

```
>>> import string
```

i zobaczmy, co w sobie kryje:

```
>>> dir(string)
['__builtins__', '__doc__', '__file__', '__name__', '_idmap', '_idmapL', '_lower', '
```

Istnieją dwie możliwości: albo zaimportujemy moduł poleceniem import nazwa_modułu, i wtedy musimy wywoływać funkcję w postacinazwa_modułu.nazwa_funkcji, albo zaimportujemy je do użycia w sposób przezroczysty, za pomocą polecenia from nazwa_modułu import nazwa_funkcji, np.:

```
>>> from string import upper
```

Możemy też od razu zaimportować wszystkie:

```
>>> from string import *
```

Wypróbujmy kilku funkcji:

```
>>> a = "Kiedy byłem mały"
>>> print upper(a)
KIEDY BYŁEM MAŁY
```

Możemy uczynić wielką pierwszą literę ciągu:

```
>>> print capitalize(a)
```

Kiedy byłem mały

Albo każdego słowa:

```
>>> print capwords(a)
Kiedy Byłem Mały
```

Uwaga: wszystkie te operacje są wykonywane na kopii łańcucha, nie na nim samym. Jeśli chcemy poddawać łańcuch dalszym modyfikacjom, najlepiej posłużyć się pomocniczymi zmiennymi. bardzo często używaną funkcją jest split(), która " rozszczepia" łańcuch znaków na wyrazy:

```
>>> b=split(a)
>>> print b[0], b[2]
Kiedy mały
```

Na której pozycji w łańcuchu znajduje się pierwszy znak ’y’?

```
>>> find(a,’y’)
4
```

Ile razy występuje w ciągu?

```
>>> count(a,’y’)
3
```

6.2 re i regex −−− wyrażenia regularne

Wyrażenia regularne pozwalają na odnalezienie znaków pasujących do podanego wzorca.

```
>>> import re
```

Znajdźmy wszystkie przypadki wystąpienia znaku ’a’:

```
>>> re.findall("a", ’Jacek Klucha’)
[’a’, ’a’]
```

...znaku ’a’ z następującym bezpośrednio po nim innym znaku:

```
>>> re.findall("a.", ’Jacek Klucha’)
[’ac’]
```

...znak ’J’ na początku linii:

```
>>> re.findall("^J", ’Jacek Klucha’)
[’J’]
```

...ciąg ’cha’ na końcu linii:

```
>>> re.findall("cha$", ’Jacek Klucha’)
[’cha’]
```

...znak ’a’, po nim znak z zakresu a−d, następnie ’e’:

```
>>> re.findall("a[a−d]e", ’Jacek Klucha’)
[’ace’]
```

Czasem użyteczne jest rozbicie ciągu na elementy przy użyciu arbitralnego ogranicznika (np. przy imporcie plików CSV):

```
>>> re.split("m", "Mamma mia")
['Ma', '', 'a ', 'ia']
```

Czasem można napotkać wykorzystanie przestarzałego modułu regex:

```
>>> import regex
>>> regex.match('a', 'ala')
1
>>> regex.match('b', 'ala')
−1
```

6.3 urllib

urllib to bardzo przydatna biblioteka do ściągania stron WWW. Ale nie tylko. Rozpatrzmy prosty przykład: ściągnięcie strony www.tpnet.pl.

```
>>> from urllib import *
```

Obiekt 'link' wskazuje na metodę open() klasy URLopener, głównej klasy zdefiniowanej w urllib.py:

```
>>> link = URLopener().open('http://www.tpnet.pl')
```

...czekamy chwilę. Jeśli wszystko poszło OK, po napisaniu

```
>>> print link
```

powinnismy otrzymać:

```
<addinfourl at 135299432 whose fp = <open file '<socket>', mode 'rb' at 80b3b20>>
```

Strona została ściągnięta, możemy się do niej dostać przy użyciustandardowej funkcji readlines:

```
>>> print link.readlines()
```

(oszczędźmy sobie wypisywania kodu HTML, który i tak będziemy zapewne chcieli przerobić).

Ale to nie wszystko. Obiekt link posiada bowiem również metodę info(), pozwalającą uzyskać informacje o połączeniu:

```
>>> print link.info()
Date: Mon, 17 Jul 2000 15:09:52 GMT
Server: Apache/1.3.9 (Unix)
Last−Modified: Thu, 20 Apr 2000 09:30:14 GMT
ETag: "2983e−1a16−38fece26"
Accept−Ranges: bytes
Content−Length: 6678
Connection: close
Content−Type: text/html
```

Możemy więc napisać proste narzędzie do badania rodzaju serwera WWW ;−). Ponieważ pole info nie jest napisem, tylko obiektem klasy Mimetools, dostajemy się do niego przy pomocy metody getheader():

```
>>> print link.info().getheader('Server')
Apache/1.3.9 (Unix)
```

A oto przykład skryptu, który wyświetli nam wszystkie tytuły wiadomości z listy dyskusyjnej LinuxPL:

```
#!/usr/bin/env python
from urllib import *
import re

a = URLopener().open("http://linux.cgs.pl/linuxpl/2000−07/index.html")
#można też urllib.urlopen()
b = a.read()
c = re.findall(r'\[linuxpl\].*.</STRONG',b)
for i in range(len(c)):
    d = c[i]
```

```
    print d[9:(len(d)-8)]
```

---

7. Programowanie obiektowe

7.1 Klasy i obiekty

Pojęcie klasy jest pojęciem abstrakcyjnym, możemy go przyrównać do przepisu na pewien produkt −−− np. opis windy. Opis windy nie jest tym samym co faktycznie realna i namacalna winda, ale czytając opis dowiemy się o przyciskach, o tym, że winda ma drzwi, o sytuacji awaryjnej. Taki opis to klasa.

Trzymając się naszej przenośni −−− obiekt to konkretnie zbudowana winda. Już nie opis, ale faktyczna stalowa klatka zawieszona na linach. W klasie mówimy Pythonowi **jak** zbudować obiekt, ale to nie klasa " działa"

a obiekt.

Mówiąc krótko −−− klasa jest abstraktem, a obiekt jej (klasy) instancją.

Metody i pola

Możemy zdefiniować (opisać) np. klasę " Kwadrat" nie odnoszącą się do żadnego konkretnego kwadratu, tylko do kwadratów w ogólności. Rozważmy poniższy skrypt:

```
#!/usr/bin/env python
# Powyżej standardowy początek skryptów w Pythonie w systemach typu Unix

# definiujemy klasę Kwadrat:
class Kwadrat:
  bok = 0

# do tego metodę podającą wartość boku
  def podaj_bok(self):
    return self.bok
```

Po kolei −−− jak widać nasza klasa zawiera jedno pole o nazwie " bok" (to tak jakby nasza winda posiadała tylko jedną cechę −−− np. drzwi). Pod tym zapisaliśmy funkcję o nazwie " podaj_bok" , która jest częścią klasy " Kwadrat" −−− funkcje składowe klasy nazywamy metodami.

W definicji metody pierwszym argumentem musi być słowo kluczowe " self" −−− powie ono nam w odniesieniu do jakiego obiektu danej klasy została wywołana metoda. Jest to także potrzebne, aby prawidłowo odnieść się do własnych pól, co możemy zobaczyć w ostatnim wierszu.

A jak to działa?

```
kw = Kwadrat()
kw.bok = 14
print kw.podaj_bok()
```

I już! Ważna uwaga −−− Python sam wstawia nasz obiekt jak argument wywoływanej metody, sami w wywołaniu już go pomijamy. Technicznie rzecz biorąc moglibyśmy się odnieść do klasy i wywołać z tego poziomu metodę explicite podając obiekt:

```
print Kwadrat.podaj_bok(kw)
```

co jest tożsame z powyższym wywołaniem, ale jest dużo mniej eleganckie i dlatego nie będziemy takiego wywołania stosować (do czasu).

Obiekty są od siebie niezależne, to tylko różne instancje tej samej klasy. Popatrzmy na poniższy kod:

```
a = Kwadrat()
b = Kwadrat()
a.bok = 14
b.bok = 300
print a.podaj_bok(), b.podaj_bok()
```

Otrzymamy dwie różne wartości −−− czego należało się oczywiście spodziewać.

Pułapki klas −−− inicjowanie pól klasy

Pułapki −−− bo przecież diabeł tkwi w szczegółach. Napisaliśmy, że klasa jest abstraktem, ale nie oznacza to, że klasa całkowicie nie może działać sama z siebie. Otóż −−− może! I ma to całkiem znaczące konsekwencje:

```
class Kwadrat:
  bok = 0
```

Czy już może domyślasz się w czym rzecz? I owszem −−− Python przemiatając kod programu uzna, iż klasa " Kwadrat" na dzień dobry przypisuje polu " bok" wartość zero. Oczywiście odbije się to na wszystkich obiektach tej klasy, ale ważne jest, że wartość pola nie zostanie ustalona w momencie

powstania obiektu:

```
kw = Kwadrat()
```

To dzieje się wcześniej −−− w momencie kiedy opisaliśmy klasę! Mimo, iż nie ma jeszcze żadnych obiektów. Czy to coś strasznego? Absolutnie nie, to wiadomość jak każda inna, należy tylko wiedzieć co się pisze −−− w przypadku pól prostych nie trzeba sobie nawet zaprzątać głowy, ale w przypadku np. list:

```
class KlasaListy:
  lista = []
```

I tutaj dokładnie wkraczamy na grząski grunt −−− ponieważ zmienna listowa jest **wskazaniem** na właściwą listę, to wszystkie obiekty tej klasy będą współużytkowały tę samą listę!!! O ile nie wykonamy dodatkowego przypisania. Oto ilustracja:

```
a = KlasaListy()
b = KlasaListy()
# dołączenie, wskazanie pozostaje niezmienione
a.lista.append("zuzia")
print a.lista, b.lista
```

Uzyskamy dwie identyczne listy. To dlatego, iż żaden z obiektów nie zmienił tego nieszczęsnego przypisania w definicji klasy. Ale tutaj:

```
a = KlasaListy()
b = KlasaListy()
# określenie nowych wskazań
a.lista = ["zuzia"]
b.lista = ["kotek"]
print a.lista, b.lista
```

---

otrzymamy dwie różne listy, dlatego, iż każdy z obiektów wykorzystuje własne przypisanie (wskazanie). Tu uwaga na marginesie −−− używanie obiektów bez zmiany wskazania jest analogiczne do pól statycznych klas w C++.

No dobrze, ale czy programista jest zmuszony do sztucznego przypisywania dla każdego z obiektów tylko po to, aby uniknąć konfliktów z innymi obiektami tej samej klasy? Nie, w takich przypadkach (tj. kiedy taki efekt jest niepożądany) należy w definicji klasy nie umieszczać przypisań.

```
class KlasaListy:
  # zupełne pominięcie <bf>inicjowania pól klasy
  def kopiuj(self,l):
    self.lista = l[:]

a = KlasaListy()
a.kopiuj(["ala"])
print a.lista
```

Inicjowanie pól obiektu

O inicjowaniu pól klasy wiemy już sporo, ale jeszcze nie wspomnieliśmy nic o inicjowaliśmy pól na poziomie obiektu, czyli o chwili, kiedy wykonywane jest np. poniższe polecenie:

```
a = KlasaListy()   # obiekt o nazwie "
a"
 rozpoczyna życie
```

Python zapewnia metodę o specjalnej nazwie " __init__" (2 podkreślenia, słowo " init" i znowu 2 podkreślenia) −−− jest ona wykonywana w momencie, kiedy przywołujemy obiekt do życia.

```
class KlasaListy:
  def __init__(self):
    print "hello, właśnie się narodziłem"

a = KlasaListy()    # niejawne wywołanie metody "
__init__"
```

"Self", czyli ten właśnie obiekt

Argument " self" wpisywany do wszystkich metodach klasy to nazwa jak każda inna −−− nie jest to słow zastrzeżone, a nazwa. Jeśli odpowiada Wam bardziej " this" nie ma żadnych przeszkód, aby tak właśnie nie pisać.

```
class KlasaListy:
  def pokaz_liste(ten_obiekt):
    print ten_obiekt.lista
```

7.2 Dziedziczenie

Klasy jako takie są już silnym narzędziem −−− pozwalają bardzo ładnie organizować kod, wiązać pewne zmienne z funkcjami, które na nich operują. Służy to dobrej organizacji i wydajniejszej konserwacji oprogramowania. Ale klasy to nie tylko pojedyncze, niezależne definicje −−− klasy można opisywać na bazie już istniejących klas.

Dodajmy do naszej klasy Kwadrat metodę obliczającą pole:

```
  def podaj_pole(self):
    return self.bok ** 2
```

Nic nadzwyczajnego prawda? Ale przecież na kwadratach świat się nie kończy −−− co jeśli zechcielibyśmy napisać klasę sześcianu? Pisać wszystko od nowa? Oczywiście, można użyć metody copy&paste w edytorze, ale Python oferuje coś więcej −−− dziedziczenie! Dziedziczenie czyli bazowanie na wskazanej klasie i rozszerzanie jej funkcjonalności w klasie pochodnej. Popatrzmy:

```
class Szescian(Kwadrat):
  ....
```

Taki zapis mówi nam, iż klasa Szescian przyjmuje cały dobytek −−− metody i pola −−− klasy Kwadrat. Wystarczyłoby dopisać:

```
  pass
```

---

i już możemy używać klasy Szescian:

```
a = Szescian()
a.bok = 10
print a.podaj_pole()
```

No tak, ale to pole kwadratu, a nie sześcianu. Co z tym fantem zrobić?

Zastępowanie metod

Musimy poprawić metodę podaj_pole. Np. definiując ją zupełnie od zera:

```
class Szescian(Kwadrat):
  def podaj_pole(self):
    return (self.bok ** 2) * 6
```

Działa to znakomicie, ponieważ Python odróżnia obiekty i w zależności od obiektu właśnie wywoła prawidłową metodę:

```
kw = Kwadrat()
sz = Szescian()
kw.bok = sz.bok = 10 # jesteśmy już hackerami prawie pełną gębą, więc możemy pozwoli
print kw.podaj_pole(), sz.podaj_pole()
```

Działa? Działa! Ale można też inaczej.

Rozszerzanie metod

Wcześniej napisaliśmy od nowa kawałek odpowiedzialny za obliczanie pola. Ale moglibyśmy też w klasie Szescian wykorzystać to, co już napisaliśmy w klasie Kwadrat:

```
class Szescian(Kwadrat):
  def podaj_pole(self):
    return ???
```

No właśnie −−− co " return" ? Nie możemy napisać " self.podaj_pole" , bo wpadniemy w nieskończoną pułapkę. Ale możemy teraz użyć opisanego wcześniej nieeleganckiego sposobu wywoływania metod −−− z podaną explicite klasą i obiektem jako argumentem:

```
    return Kwadrat.podaj_pole(self) * 6
```

I to jest to! Specyfikując klasę przed nazwą metody zmuszamy Pythona, aby wywołał metodę dokładnie wg zapisu w tej właśnie klasie. Ale klasa jest abstraktem, klasa nie pracuje! I dlatego podajemy jako argument " self" −−− jest to obiekt typu Szescian, ale dziedziczy on po Kwadracie. Czyli zmuszamy obiekt pochodny do udawania przez chwilę obiektu podstawowego −−− albo inaczej −−− zmuszamy klasę podstawową, aby jej metody przetworzyły obiekt pochodny. W tę stronę ta sztuczka się udaje, ponieważ klasa pochodna zawiera wszystkie składowe klasy podstawowej −−− w drugą stronę jednak to nie przejdzie.

Wiązanie metod do obiektu

Python wiąże metody klasy z obiektem, do klasy którego należą −−− niezależnie od miejsca wywołania. Jeśli metoda nie istnieje (a może), nastąpi przejście do klasy poprzednika. Ale tylko na czas wyszukiwania tej jednej metody. Przy następnym przeszukiwaniu proces znowu rozpocznie się od klasy obiektu. Popatrzmy:

```
class Kwadrat:
  bok = 0
  def wyswietl(self):
    print "bok:", self.bok, "; pole:", self.pole()
  def pole(self):
    return self.bok ** 2

class Szescian(Kwadrat):
  def pole(self):
    return Kwadrat.pole(self) * 6

sz = Szescian()
sz.bok = 2
sz.wyswietl()
```

Wynik działania −− 24, a więc prawidłowy. Jak to wszystko przebiegło? Interesuje nas punkt począwszy od wywołania metody " wyswietl" :

1. obiekt " sz" jest klasy " Szescian" i tutaj Python rozpocznie szukanie metody " wyswietl" ,
2. klasa " Szescian" nie zawiera takiej metody, brany jest w takim przypadku poprzednik −−− " Kwadrat" ,
3. ta klasa zawiera żądaną metodę −−− generowane jest wywołanie " Kwadrat.wyswietl(sz)" ,
4. metoda " wyswietl" klasy " Kwadrat" wywołuje z kolei metodę " pole" ,
5. rzecz tyczy cały czas obiektu klasy " Szescian" (mimo, iż obecnie znajdujemy się w obrębie klasy " Kwadrat") −−− przeszukiwana jest klasa " Szescian" i metoda " pole" zostaje znaleziona,
6. metoda " pole" klasy " Szescian" z kolei wywołuje explicite metodę klasy " Kwadrat" ,
7. zostaje ona odnaleziona, wykonana i cały proces odwija się do wywołania, od którego zaczęliśmy.

7.3 Konkretny przypadek

Wśród modułów dostarczanych razem z Pythonem znajduje się ConfigParser. Nie jest to moduł

rewelacyjny, ale łatwo się z niego korzysta. Służy on do odczytu opcji z plików konfiguracyjnych (podzielonych na sekcje, każdy nagłówek sekcji znajduje się w nawiasach kwadratowych, każdej opcji przypisana jest jakaś wartość, która jest oddzielona od tej pierwszej znakiem '=' lub ':').)Dla naszych potrzeb trzeba będzie zmodyfikować nieco jedną linijkę, ponieważ wersja ortodoksyjna nie toleruje białych znaków w nagłówkach sekcji.

Dlatego w /usr/lib/python1.5/ znajdujemy plik ConfigParser.py i kopiujemy go do swojego katalogu, w którym eksperymentujemy. Na początku skryptu po linijce

---

```
#!/usr/bin/env python
```

---

dodajemy

```
import ConfigParser
```

W samym pliku zamieniamy (u mnie jest to linijka 327) kompilowane wyrażenie regularne __SECTCRE z

---

```
...
r'(?P<header>[-\w]+)'
...
```

---

na

```
...
r'(?P<header>[-\w \t]+)'.
...
```

Dalsza część tekstu zakłada, że czasem używamy KDE. A to z tego powodu, że .kderc jest jedynym plikiem w moim katalogu domowym, który z grubsza przypomina modelowy plik konfiguracyjny (obecność sekcji itd.).

W pliku ConfigParser jest zadeklarowana klasa ConfigParser −− znajduje się tam również krótki opis metod tej klasy. Metody są funkcjami właściwymi dla danej klasy. Aby skorzystać z owych metod, musimy najpierw utworzyć obiekt należący do klasy ConfigParser (zadeklarowanej w pliku o tej samej

nazwie −− stąd powtórzenie):

```
a = ConfigParser.ConfigParser()
```

Możemy teraz zobaczyć, jakie narzędzia mamy do dyspozycji:

```
>>> dir(ConfigParser.ConfigParser)
['_ConfigParser__OPTCRE', '_ConfigParser__SECTCRE', '_ConfigParser__get', '_ConfigPa
```

Najpierw wywołamy metodę read(), która wczyta plik konfiguracyjny:

```
>>> a.read('.kderc')
```

Następnie sprawdźmy, jakie nagłówki zawiera nasz plik .kderc:

```
>>> print a.sections()
['General', 'KDE', 'Standard Keys', 'WM', 'KFileDialogSettings', 'Locale']
```

Aby sprawdzić wartości, używamy metody get(). Sprawdźmy, jak się jej używa:

```
>>> print a.get.__doc__
Get an option value for a given section.

        All % interpolations are expanded in the return values, based on the
        defaults passed into the constructor, unless the optional argument
        'raw' is true.  Additional substitutions may be provided using the
        'vars' argument, which must be a dictionary whose contents overrides
        any pre-existing defaults.

        The section DEFAULT is special.
```

Spróbujemy więc odczytać wartość opcji " Next" z sekcji " WM" :

```
>>> print a.get("WM","Next", raw=0, vars=None)
PageDown
```

8. Wyjątki −−− bezpieczne programowanie

" Wyjątki" to skrót myślowy od " sytuacje wyjątkowe" , a mniej eufemistycznie −−− błąd w wykonywaniu (a nie prekompilowaniu) programu. Wyjątki to mechanizm nie tyle zapobieganiu groźnym sytuacjom, ale sposób ich zgrabnego opanowania bez palpitacji serca u użytkownika.

Zanim zaczniemy opisywać głębiej sam temat, parę słów pochwały −−− mechanizm wyjątków autentycznie zmienił oblicze programowania. Dzięki niemu można zapisać kod w sposób klarowniejszy, bardziej usystematyzowany, a w efekcie użytkownik powinien otrzymać oprogramowanie dużo stabilniejsze. Bez mechanizmu wyjątków to samo było oczywiście możliwe po stronie użytkownika, ale programista musiał nieźle zużyć klawiaturę, zanim doszedł do tego samego, o czym za chwilę opowiemy.

8.1 Prehistoria

Popatrzmy na banalny kod:

```
x = 10
y = 0
z = x / y
```

Wykonanie takiego programu się nie powiedzie, to rzecz jasna −−− możemy przeciwdziałać błędom np. tak:

```
if y == 0:
  print "Dzielnik równy zeru; niepoprawne dane"
else:
  z = x / y
```

Tutaj nawet nie za bardzo widać ewentualnej potrzeby korzystania z wyjątków, w końcu te dwie linijki więcej... Ale jeśli danych jest więcej −−− np. zbieramy od użytkownika informacje potrzebne do zapisu do bazy danych −−− to chciałoby się nam bawić w dziesiątki " if" ? Raczej nie. O wiele wygodniej puścić program na żywioł −−− a niech sobie tam dzieli co chce, konwertuje ile wlezie −−− a jeśli wystąp błąd obsłużymy taką sytuację zbiorczo. Zapiszmy to schematycznie tak:

```
zbierz_dane()
blad = interpretuj_dane()
if blad:
  print "Wprowadziłeś niepoprawne dane"
zapisz_dane()
```

8.2 Konstrukcje obsługi wyjątków

Na marginesie: jeśli znacie Object Pascala, poczujecie się jak w domu −−− nazewnictwo i konwencja działania jest prawie identyczna.

try−except−else (przechwytywanie definitywne)

```
try:
  ...  # kod programu
except:
  ...  # kod wykonywany w przypadku błędu
```

Python wykonuje kod zawarty między klauzulami "try−except" i jeśli zdarzy się tu błąd wykonywanie przeskoczy od razu do bloku " except" . Dlaczego jest to przechwycenie definitywne −−− ewentualny błąd nie wycieka poza blok " except" , jest tam przechwytywany i zabijany.

Oto nowa wersje dzielenia:

```
try:
  z = x / y
except:
  print "Niepoprawne dane"
```

Dodatkowo możemy dopisać sekcję " else" −−− zostanie ona wykonana kiedy blok " try−except" zostani wykonany bez żadnego błędu:

```
try:
  z = x / y
except:
  print "Niepoprawne dane"
else:
  print "Wynik:", z
```

Zwróć uwagę, że ostatni wiersz zostanie wykonany tylko i wyłącznie w przypadku braku błędu, gdybyś zapisał to tak:

```
try:
  z = x / y
except:
  print "Niepoprawne dane"

print "Wynik:", z
```

To kod taki (wyświetlenie wyniku) zostałby wykonany w obu przypadkach!

try−finally (pośrednio i obligatoryjnie)

Konstrukcję tę zapisujemy analogicznie do " try−except" , ale jej działanie różni się w dwóch punktach:

1. sekcja " finally" będzie wykonana zawsze! Jeśli błąd wystąpi, to zaraz po jego wystąpieniu, jeśli nie −−− po ostatniej linii " try" zostanie wykonana pierwsza linia " finally"
2. sekcja " finally" przekazuje ewentualne błędy dalej −−− propagacja nie zostaje zatrzymana

```
# y określamy wcześniej
x = 10
try:
  z = x / y
finally:
  print "koniec"
```

Łatwo się przekonamy, że dla " y" równego zero, zobaczymy napis " koniec" , ale zobaczymy także, że Python się nas obraził przerywając program (" propagacj nie zostaje zatrzymana" −−− to się sprawdza), natomiast dla " y" równego powiedzmy jeden, także zobaczymy napis (ale program będzie działał dalej).

W takim razie −−− czy to ma sens? I owszem, bowiem " try−except"

ma zapobiegać wyciekaniu błędów, ma dawać użytkownikowi informacje, a " try−finally" ma pomagać programiście w zrobieniu porządku w danych. Oto klasyczny przykład (w odcinkach):

```
try:
  plik = open("zuzia.txt",'r')
  s = plik.readline()
```

```
  print s
finally:
  plik.close()
```

Taki kod zabezpiecza nas całkiem ładnie na wypadek błędów podczas czytania pliku −−− jeśli cokolwiek złego się wydarzy nastąpi przeskok do sekcji " finally" i plik zostanie zamknięty. Jeśli wszystko przebiegło zgodnie z planem −−− sekcja " finally" także zostanie wykonana, a plik zamknięty. Ale jest jeden szkopuł −−− jeśli pliku w ogóle nie będzie, zmienna " plik" będzie niezdefiniowana i w sekcji " finally"

wystąpi dodatkowy błąd! Możemy temu zaradzić dopisując:

```
plik = 0
try:
  ...
finally:
  if plik:
    plik.close()
```

Teraz lepiej −−− jeśli plik nie zostanie otworzony, sprawdzimy to i nie będziemy go zamykać. Ale to nie wszystko −−− błąd przecież się propaguje, przerywając program. Złu zaradzimy dodając blok " try−except"

−−− oto kompletny kod:

```
plik = 0
try:
  try:
    plik = open("zuzia.txt",'r')
    s = plik.readline()
    print s
  finally:
    if plik:
      plik.close()
except:
  print "Nastąpił błąd podczas czytania pliku"
```

Zamknięcie pliku zostanie wykonane zawsze (oprócz sytuacji, kiedy dany plik w ogóle nie istnieje), także wtedy, kiedy wszystko przebiegło bez przeszkód, natomiast komunikat użytkownik zobaczy tylko wtedy, kiedy faktycznie wystąpił błąd −−− czy to błąd otwarcia, czy czytania. Program jako taki nie zostanie przerwany.

I o to chodzi, i to chodzi.

Tajniki "try−finally"

" Finally" w Pythonie naprawdę znaczy, to co znaczy −−− akcję finałową. Ponieważ jak już wiemy, sekcja ta dotyczy nie tylko sytuacji wyjątkowych, ale obejmuje ogół (wykonywana jest zawsze) −−− ma ona wpływ także na klasyczne polecenia:

```
def pomnoz(x,y):
  try:
    return x*y
  finally:
    print "Wynik:",

print pomnoz(3,4)
```

Jeśli spodziewacie się zobaczenia napisu " Wynik" to intuicja Was nie zwodzi −−− rzeczywiście, mimo, iż polecenie " return" natychmiast opuszcza funkcje, to musi po drodze odwiedzić sekcję " finally" −−− " finally" ma znaczenie nadrzędne!

Rozszerzone "except"

" Except" oprócz budowy takiej, jaką poznaliśmy do tej pory pozwala także nam na rozróżnianie typów błędów wraz z towarzyszącym komunikatem. Wróćmy do naszego przykładu z dzieleniem przez zero:

```
x = 10
y = 0
try:
  z = x / y
except ZeroDivisionError, komunikat:
  print komunikat
```

Przechwytujemy nie jakiś tam wyjątek, ale dokładnie określony −−− w rzeczywistych jednak programach, lepiej nie pozostawiać dziur na sytuacje inne niż przewidzieliśmy pisząc program. Rozsądnie jest dopisać jeszcze dwa wiersze:

```
except:
  print "Inny wyjątek"
```

Najczęściej jednak będziemy łączyć taki zapis i korzystać z wyjątku ogólnego typu (tj. klasy podstawowej wyjątków, ponieważ wyjątki to klasy):

```
try:
  z = x / y
except Exception, komunikat:
  print komunikat
```

8.3 Generowanie wyjątków

raise ("...i powstał wyjątek")

Wywoływanie wyjątków daje nam znakomitą możliwość zintegrowania się z wewnętrznymi mechanizmami Pythona i ujednolicenie programu, jeśli chodzi o komunikację z użytkownikiem i zapis samego kodu.

Przykład może nie najambitniejszy:

```
plik = 0

try:
  plik = open("dni.txt",'r')
  while 1:
    s = plik.readline()
    if not s:
      break
    dzien = eval(s[:-1]) # pomijamy znak nowego wiersza
    if not (1 <= dzien <= 31):
    raise Exception, "dzień spoza zakresu"

except Exception, komunikat:
  print "Błąd:", komunikat

if plik:
  plik.close()
```

Polecenie "raise", bo ono jest tu nowinką, generuje wyjątek typu podanego w pierwszym parametrze, z komunikatem określonym w drugim (nieobowiązkowym) parametrze.

Dzięki takiemu zapisowi mamy spójny kod, który zabezpiecza nas przed brakiem pliku, błędami natury technicznej w odczycie, niepoprawnymi danymi w samej treści −− użytkownik dostaje komunikat, który coś znaczy, a przy tym program nie przerywa działania i sprząta po sobie (zamknięcie otwartego pliku).

Własne typy wyjątków

Jeżeli chcemy dokładnie usystematyzować wyjątki, możemy zamiast posługiwać się ogólnym typem, zdefiniować swój własny −−− należy tylko pamiętać, że podstawowym typem wszystkich wyjątków

powinna być klasa "Exception":

```
class ExceptionNiepoprawnyDzien(Exception):
  pass

...
    if not (1 <= dzien <= 31):
      raise ExceptionNiepoprawnyDzien, "dzień spoza zakresu"
...
```

Oczywiście i w tym przypadku parametr komunikatu moglibyśmy opuścić.

Dlaczego nie możemy definiować własnych wyjątków na przykład tak

---

```
MojWyjatek = "mój wyjątek" # tekstowy wyróżnik wyjątku

try:
  raise MojWyjatek
except MojWyjatek:
  print "Błąd"
```

...tym bardziej, jeśli to działa? No tak −−− działa −−− ale do czasu. Jeśli chcielibyśmy reagować na wyjątki w sposób ogólny:

```
except Exception:
  print "Błąd"
```

---

to nasz własny wyjątek prześlizgnie się przez sieci −−− nie należy on do klasy (tj. do drzewa klas) typu "Exception".

Podsumowując −−− zapomnij o tym sposobie!

Raise −−− wyjątki w sztafecie

Jeżeli to potrzebne możemy z sekcji "except" wywołać wyjątek właśnie przechwycony albo inaczej mówiąc −−− pozwolić mu przejść dalej, tak jak to dzieje się w sekcji "finally". Ponieważ jest to dokładnie ten sam wyjątek, który złapaliśmy nie podajemy żadnych parametrów:

```
try:
  z = x / y
except:
  print "Błąd dzielenia"
  raise
```

Czy wygodniejsze jest użycie "finally", czy "except−raise" zależy od programisty i konkretnej sytuacji −−− Python w każdym razie pozostawia szerokie pole manewru.

9. Czyżby koniec?

Chętni zainteresowani rozwinięciem materiału o Pythonie proszeni są o kontakt ;−)