Kurs języka Python

Wykład 4.

Marcin Młotkowski

26 października 2009

1. Uzupełnienie
2. Programowanie obiektowe
3. Wyjątki
4. Zmienne
5. Metody specjalne

Uzupełnienie — listy

Implementacja list

Wektor wskaźników

Złożoność operacji

Czas dostępu: $O(1)$
Wstawianie/usuwanie elementów:

- na końcu: zamortyzowany czas $O(1)$
- poza tym: $O(n)$

Specjalizowane listy

Na przykład `collections.deque`; wstawianie i usuwanie z obu końców: $O(1)$

Listy składane contra użycie funkcji

Przykłady

```
def skladana():
    return [ x for x in range(10) if x % 7 == 0]

def zwykla():
    return filter(lambda x: x % 7 == 0, range(10))
```

Rezultat

Funkcja filter 4.68412303925
Lista skladana 2.93897509575

Listy składane contra użycie funkcji

Przykłady

```
def skladana():
    return [ x for x in range(10) if x % 7 == 0]

def zwykla():
    return filter(lambda x: x % 7 == 0, range(10))
```

Rezultat

Funkcja filter 4.68412303925
Lista skladana 2.93897509575

Lambda wyrażenia

Budowa lambda wyrażenia

lambda x: wyrażenie

Definicja klasy

Przykłady

```
class Figura:
    """Pierwsza klasa"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Definicja metody

class Figura, cd. definicji

```
    ...
    def info(self):
        print self.x, self.y

    def zmien(self, x, y):
        self.x = x
        self.y = y
```

Tworzenie obiektów i wywołanie metod

Przykład

```
o = Figura(1, -1)
o.info()
o.zmien(2,3)
o.info()
```

Dziedziczenie

```
class Okrag(Figura):
    """Okrag"""
    def __init__(self):
        self.x, self.y, self.r = 0, 0, 1

    def info(self):
        print 'x = %i, y = %i, r = %i', (self.x, self.y, self.r)
```

Wywołanie konstruktora z nadklasy

```
    def __init__(self):
        Figura.__init__(self)
        ...
```

Metody wirtualne

Class Figura

```
def info(self):
    ...
def przesun(self, dx, dy):
    self.info()
    self.x, self.y = self.x + dx, self.y + dy
    self.info()
```

```
okrag = Okrag();
okrag.przesun(10,15)
```

Metody wirtualne

Class Figura

```
def info(self):
    ...
def przesun(self, dx, dy):
    self.info()
    self.x, self.y = self.x + dx, self.y + dy
    self.info()
```

```
okrag = Okrag();
okrag.przesun(10,15)
```

Wielodziedziczenie

```
class Samochod:
    def naprzod(self):
```

```
class Okret:
    def naprzod(self):
```

```
class Amfibia(Samochod, Okret):
```

Zagadka

```
amf = Amfibia()
amf.naprzod()
```

Wielodziedziczenie

```
class Samochod:
    def naprzod(self):
```

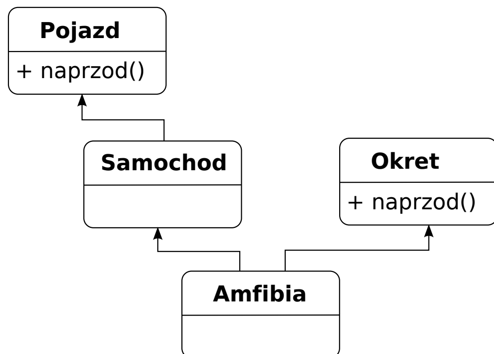
```
class Okret:
    def naprzod(self):
```

```
class Amfibia(Samochod, Okret):
```

Zagadka

```
amf = Amfibia()
amf.naprzod()
```

Rozwiązywanie konfliktów



Reguła

W głąb, od lewej do prawej

Wyjątki

- Mechanizm przepływu sterowania
- Wyjątki to obiekty

Wyjątki

- Mechanizm przepływu sterowania
- Wyjątki to obiekty

Obsługa wyjątków

```
try:
    f = open("plik"[10] + ".py", "r")
except IOError
    print "Błąd wejścia/wyjścia"
except IndexError as x:
    print x
except:
    print "Nieznany wyjątek"
finally:
    f.close()    # Dopiero od 2.5!
```

Obsługa wyjątków

```
try:
    f = open("plik"[10] + ".py", "r")
except IOError
    print "Błąd wejścia/wyjścia"
except IndexError, x:
    print x
except:
    print "Nieznany wyjątek"
finally:
    f.close()     # Dopiero od 2.5!
```

Klauzula else

```
try:
    print 2/n
except:
    print "Nieudane dzielenie"
else:
    print "Udane dzielenie"
```

Zgłaszanie wyjątków

raise

```
raise RuntimeError("Stało się coś złego")

raise "Wyjątek"
```

Zgłaszanie wyjątków

raise

```
raise RuntimeError("Stało się coś złego")

raise "Wyjątek"
```

Zmienne, podstawowe fakty

Fakt 1

Zmienne tworzone są w momencie przypisania

```
x = 1
```

```
class Klasa:
    def __init__(self):
        self.y = 12
```

Zmienne, podstawowe fakty

Fakt 1

Zmienne tworzone są w momencie przypisania

```
x = 1
```

```
class Klasa:
    def __init__(self):
        self.y = 12
```

Zmienne, podstawowe fakty

Fakt 1

Zmienne tworzone są w momencie przypisania

```
x = 1
```

```
class Klasa:
    def __init__(self):
        self.y = 12
```

Zmienne, podstawowe fakty

Fakt 1

Zmienne tworzone są w momencie przypisania

```
x = 1
```

```
class Klasa:
    def __init__(self):
        self.y = 12
```

Widzialność zmiennych

Fakt 2

Zmienne są widziane tylko w miejscu, w którym są utworzone (tj. w funkcji czy module)

Zmienne lokalne funkcji i metod

```
x = "jeden"
def foo():
    x = 1
```

Widzialność zmiennych

Fakt 2

Zmienne są widziane tylko w miejscu, w którym są utworzone (tj. w funkcji czy module)

Zmienne lokalne funkcji i metod

```
x = "jeden"
def foo():
    x = 1
```

Zmienne lokalne modułu

modul.py

```
zmienna_modulu = 10
```

plik.py

```
import modul
print modul.zmienna_modulu
```

Zmienne lokalne modułu

modul.py

```
zmienna_modulu = 10
```

plik.py

```
import modul
print modul.zmienna_modulu
```

Pola statyczne klasy

```
class Okrag:
    pi = 3.1415
    def __init__(self):
        self.r = 2.71
    def pole(self):
        print "Pole okręgu = %i"  %  (Okrag.pi * self.r **2 )
```

Odwołanie do pól statycznych klasy

```
print Okrag.pi

o = Okrag()
print o.pi
```

Pola statyczne klasy

```
class Okrag:
    pi = 3.1415
    def __init__(self):
        self.r = 2.71
    def pole(self):
        print "Pole okręgu = %i"  %  (Okrag.pi * self.r **2 )
```

Odwołanie do pól statycznych klasy

```
print Okrag.pi

o = Okrag()
print o.pi
```

Pola obiektu

```
class Okrag:
    pi = 3.1415
    self.x, self.y = 0, 0
    def __init__(self):
        self.x, self.y = 0, 0
```

Zmienne

Fakt 3.

Zmienne można dodawać dynamicznie

Nowa zmienna modułu

```
modul.nowa_zmienna = 'Nowa zmienna'
```

Nowa zmienna obiektu

```
o = Figura()
o.nowe_pole = "Nowe pole"
```

Zmienne

Fakt 3.

Zmienne można dodawać dynamicznie

Nowa zmienna modułu

```
modul.nowa_zmienna = 'Nowa zmienna'
```

Nowa zmienna obiektu

```
o = Figura()
o.nowe_pole = "Nowe pole"
```

Zmienne

Fakt 3.

Zmienne można dodawać dynamicznie

Nowa zmienna modułu

```
modul.nowa_zmienna = 'Nowa zmienna'
```

Nowa zmienna obiektu

```
o = Figura()
o.nowe_pole = "Nowe pole"
```

Zmienne

Fakt 4.

Zmienne można usuwać dynamicznie

Przykład

```
x = 'x'
del x
```

Zmienne

Fakt 4.

Zmienne można usuwać dynamicznie

Przykład

```
x = 'x'
del x
```

Zmienne globalne

```
zmienna = 'wartość'
def foo():
    global zmienna # deklaruje, ale nie tworzy!
    print zmienna # tworzy zmienną
```

Zmienne prywatne

Zmienną prywatną jest zmienna poprzedzona dwoma podkreśleniami i zakończona co najwyżej jednym podkreśleniem (dotyczy modułów i klas).

Np.

__zmiennaPrywatna

Zmienne specjalne — przykłady

Klasa obiektu

```
>>>"Napis".__class__
<type 'str'>
```

Dokumentacja klasy

```
Figura.__doc__
Figura.__dict__
```

```
plik.__file__
```

Nazwa modułu

```
__name__
```

Zmienne specjalne — przykłady

Klasa obiektu

```
>>>"Napis".__class__
<type 'str'>
```

Dokumentacja klasy

```
Figura.__doc__
Figura.__dict__
```

```
plik.__file__
```

Nazwa modułu

```
__name__
```

Zmienne specjalne — przykłady

Klasa obiektu

```
>>>"Napis".__class__
<type 'str'>
```

Dokumentacja klasy

```
Figura.__doc__
Figura.__dict__
```

```
plik.__file__
```

Nazwa modułu

```
__name__
```

Zmienne specjalne — przykłady

Klasa obiektu

```
>>>"Napis".__class__
<type 'str'>
```

Dokumentacja klasy

```
Figura.__doc__
Figura.__dict__
```

```
plik.__file__
```

Nazwa modułu

```
__name__
```

Implementacja kolekcji

Przetwarzanie kolekcji

```
for element in kolekcja: print x
```

Mechanizm

- Na początku wywoływana jest metoda __iter__
- Wartością powinien być obiekt (enumerator) implementujący metodę next() która za każdym wywołaniem zwraca kolejny element kolekcji
- Metoda next() jest wywoływana tak długo, póki nie zostanie zgłoszony wyjątek StopIteration

Implementacja kolekcji

Przetwarzanie kolekcji

```
for element in kolekcja: print x
```

Mechanizm

- Na początku wywoływana jest metoda __iter__
- Wartością powinien być obiekt (enumerator) implementujący metodę next() która za każdym wywołaniem zwraca kolejny element kolekcji
- Metoda next() jest wywoływana tak długo, póki nie zostanie zgłoszony wyjątek StopIteration

Implementacja kolekcji

Przetwarzanie kolekcji

```
for element in kolekcja: print x
```

Mechanizm

- Na początku wywoływana jest metoda __iter__
- Wartością powinien być obiekt (enumerator) implementujący metodę next() która za każdym wywołaniem zwraca kolejny element kolekcji
- Metoda next() jest wywoływana tak długo, póki nie zostanie zgłoszony wyjątek StopIteration

Implementacja kolekcji

Przetwarzanie kolekcji

```
for element in kolekcja: print x
```

Mechanizm

- Na początku wywoływana jest metoda __iter__
- Wartością powinien być obiekt (enumerator) implementujący metodę next() która za każdym wywołaniem zwraca kolejny element kolekcji
- Metoda next() jest wywoływana tak długo, póki nie zostanie zgłoszony wyjątek StopIteration

Przykład

Zadanie

Implementacja kolekcji zwracającej kolejne liczby od 1 do 10

Implementacja

```
class ListaLiczb:
    def __iter__(self):
        self.licznik = 0
        return self
    def next(self):
        if self.licznik >= 10:  raise StopIteration
        self.licznik += 1
        return self.licznik
```

Przykład

Zadanie

Implementacja kolekcji zwracającej kolejne liczby od 1 do 10

Implementacja

```
class ListaLiczb:
    def __iter__(self):
        self.licznik = 0
        return self
    def next(self):
        if self.licznik >= 10: raise StopIteration
        self.licznik += 1
        return self.licznik
```