Programmierkurs Python I – WS 09/10

Übung 9

---

1 Dekoratoren als Typ-Tester (2 Punkte)

Schreibe einen Dekorator der testet, ob die Parameter einer Funktion beim Aufruf einen bestimmten Typ haben. Die Funktion soll nur dann ausgeführt werden, wenn die Parameter-Typen akzeptiert werden. Als Parameter soll der Dekorator die Argument-Typen bekommen.

```
@check(str,int,list)
def eineMethode(wort, ganze_zahl, liste):
       # der Code hier soll nur ausgeführt werden, wenn wort ein
       # String, ganze_zahl ein int und liste ein list-Objekt ist
```

Der Dekorator soll so generisch sein, dass es egal ist, wie die zu dekorierende Methode aussieht und wie viele Parameter sie bekommt.

Hinweis: Ob ein Objekt `obj` einen bestimmten Typ `t` hat, überprüft man mit `isinstance(obj,t)` (siehe Übung 8).

2 List Comprehensions (2 Punkte)

Schreibe eine Funktion `prime_list(n)`, die die Primzahlen von 1 bis n in einer Liste zurückgibt. Die Rückgabeliste soll nur über (möglichst wenige) List Comprehensions definiert werden.

3 Singletons (3 Punkte)

Die Singleton-Definition aus der Vorlesung erfüllt eigentlich nicht alle Bedingungen für Singletons. Eigentlich sollte die Implementierung Sicherstellen, dass nur eine einzige Instanz erzeugt werden kann. Mit der Beispielklasse können aber beliebig viele Instanzen erzeugt werden, wenn der Konstruktor (`Singleton()`, bzw. die abgeleitete Klasse) direkt aufgerufen wird. Skizziere (in Worten), was man tun müsste, damit wirklich nur eine einzige Instanz einer Klasse erzeugt werden kann.

4 Dekoratoren als Tracer (Bonus, 3 Punkte)

Schreibe einen Dekorator für rekursive Funktionen, der als Debug-Ausgabe den aktuellen Funktionsaufruf mit Parameter-Belegung und das Ergebnis des Funktionsaufrufes ausgibt. Aus der Ausgabe soll hervorgehen, wie die rekursiven Aufrufe verschachtelt sind und welcher Funktionsaufruf zu welchem Ergebnis gehört. Im Beispiel sind Funktionsaufrufe mit `Call` gekennzeichnet, Ergebnisse mit `Result`, und die Rekursionstiefe mit Einrückungen visualisiert und die Zusammengehörigkeit von Aufruf und Ergebnis der Funktion mit Nummern gekennzeichnet:

```
>>> @trace('fib')
>>> def fib(n):
...       return n if n < 2 else fib(n - 1) + fib(n - 2)
...
>>> fib(3)
Call[1]: fib(3)
  Call[2]: fib(2)
    Call[3]: fib(1)
    Result[3]: 1
    Call[4]: fib(0)
    Result[4]: 0
  Result[2]: 1
  Call[5]: fib(1)
  Result[5]: 1
Result[1]: 2
2
```

5 `map`, `lambda` (Bonus, 3 Punkte)

Schreibe einen Ausdruck, der aus zwei Listen von 2-Tupeln eine neue Liste mit Funktionen berechnet, so dass

- jedes 2-Tupel als ein Punkt angesehen wird
- jeweils 2 Punkte an der gleichen Listenposition zusammen gehören
- in der endgültigen Liste die Funktionen stehen, die bestimmen, ob ein Punkt auf der Geraden durch die zwei korrespondierenden Listenpunkte geht.

```
>>> l1 = [(1,1), (2,3), ...]
>>> l2 = [(3,3), (1,4),...]
>>> # evtl. Code
>>> function_list = # euer Code
>>> # evtl. Code
>>> function_list[0]((0,0))
True
```

Benutze `map` für die Listenerzeugung und `lambda` für das Erzeugen der Funktion, die in `map` übergeben wird. Wie man eine Geradengleichung mit zwei Punkten aufstellt, steht unter `http://de.wikipedia.org/wiki/Geradengleichung`.

Wie ist Deine Meinung über den Code hinsichtlich des Programmierstil?

Abgabe bis Donnerstag, 28.01.2010, 14:00 Uhr