PYTHON

Chapter: Functions

Functions – defining and calling

```
>>> def sayhi():
        print "Hi“

>>> type(sayhi)
<type 'function'>
>>> sayhi()
Hi
```

```
>>> def sayhi():
        ''' this is a function
that says hi '''
        print "Hi"


>>> sayhi()
Hi
>>> sayhi.__doc__
' this is a function that says hi
'
```

Functions

```
>>> def is_even(num):
        if num%2 == 0:
                return True
        else:
                return False
>>> if is_even(2):
        print "This is even"
else:
        print "This is odd"


This is even
```

```
if is_even(3):
        print "This is even"
else:
        print "This is odd"


This is odd
```

Functions – Parameters (key word)

```
>>> def full_name(fname, lname):
       print fname + " " + lname


>>> full_name('Aditya', 'Prabhakara')
Aditya Prabhakara
>>> full_name (lname='Prabhakara', fname='Aditya')
Aditya Prabhakara
```

Functions – Parameters (default args)

```
>>> def full_name(fname, lname, title="Mr"):
       print title + " " + fname + " " + lname


>>> full_name (lname='Prabhakara', fname='Aditya')
Mr Aditya Prabhakara
>>> full_name(lname='Prabhakara')

Traceback (most recent call last):
  File "<pyshell#785>", line 1, in <module>
    full_name(lname='Prabhakara')
TypeError: full_name() takes at least 2 arguments (1 given)
```

Functions – any number of args -1

```
>>> def any_args(*args):
        print args


>>> any_args('a',1,2,'c')
('a', 1, 2, 'c')
>>> def any_args(*args):
        print args
        print type(args)


>>> any_args('a',1,2,'c')
('a', 1, 2, 'c')
<type 'tuple'>
```

PYTHON

Functions – any number of args - 2

```
>>> def any_args(**kwargs):
       print kwargs

>>> any_args(what=2, where=3)
{'what': 2, 'where': 3}

>>> def any_args(*args, **kwargs):
       print args, kwargs



>>> any_args(1, 2, 3,a=4, b = 5, c = 6)
(1, 2, 3) {'a': 4, 'c': 6, 'b': 5}
```

Chapter: Functions as First class citizens

Functions (Contd)

- Functions are first class citizens
- Which basically means
    - They are treated like objects (as in Python everything is an object)
    - They can be passed around like other variables
    - Assigned to other variables

PYTHON

Function as variables

```
>>> def sayhi():
        print "Hi"


>>> sayhi()
Hi
>>> type(sayhi)
<type 'function'>
>>> a = sayhi
>>> type(a)
<type 'function'>
>>> id(a)
59704264L
>>> id(sayhi)
59704264L
```

```
>>> a()
Hi
>>> def someotherhi(func):
        func()


>>> someotherhi(a)
Hi
>>>
```

Inner functions

```
>>> def add(*args):
        def inneradd(*args):
                return sum(*args)
        print inneradd(*args)



>>> add([1,2])
3
>>> add([1,2,3,4])
10
>>> sum.__doc__
"sum(sequence[, start]) -> value\n\nReturn the sum of a sequence of
numbers (NOT strings) plus the value\nof parameter 'start' (which
defaults to 0).  When the sequence is\nempty, return start."
```

Chapter: Comprehension

range

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

Comprehension

- Comprehension is a compact way of creating a Python data structure
- Leads to more readable code
- List comprehension is a compact way of creating lists
- A generic syntax of comprehension is as follows

```
[expression for item in iterable]
```

- What qualifies as iterable?
    - Lists
    - Range
    - Generators

  … and a lot more

List comprehension

```
>>> a = [x for x in range(5)]
>>> a
[0, 1, 2, 3, 4]
>>> a = [x*x for x in range(5)]
>>> a
[0, 1, 4, 9, 16]
>>> b = 'abcdefghijklmnopqrstuvwxyz'
>>> a = [x*2 for x in b]
>>> def sqit(x):
        return x*x

>>> a = [sqit(x) for x in range(5)]
>>> a
[0, 1, 4, 9, 16]
```

PYTHON

List comprehension – advanced

```
[expression for item in iterable if expression]
```

```
>>> a = [x for x in range(5) if x%2 == 0]
>>> a
[0, 2, 4]

>>> a = [x for x in range(5) if is_even(x)]
>>> a
[0, 2, 4]
```

Dict comprehension

```
{keyexpr:valueexpr for item in iterable if expression}
```

```
>>> a = {x:x*x for x in range(5)}
>>> a
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# comprehend and deliberate on what is happening below

>>> a = 'the quick brown fox jumps over the lazy dog'
>>> a={x:a.count(x) for x in a}
```

Chapter: Lambda

Lambda

- Anonymous functions
- Expressed as a single statement
- Use it instead of a normal tiny functions
- Can be used as closures

PYTHON

Lambda

- Import only what you like
- Some hate it. Some love it. But you can’t ignore it ☺

```
>>> def f(x):
       return x%2==0

>>> f(1)
False
>>> f(2)
True
>>> d = lambda x:x%2==0
>>> d(5)
False
>>> d(4)
True
```

Map

- Takes a function and applies it to every item in the list

```
>>> raceinkm =[5,10,21,42]
>>> def kmtomi(x):
        return x*0.621

>>> map(kmtomi, raceinkm)
[3.105, 6.21, 13.041, 26.082]
>>> map(lambda x:x*0.621, raceinkm)
[3.105, 6.21, 13.041, 26.082]
>>> salary =[1.4,0.8,2.6,5.8]
>>> bonus=[0.2,.10,.40,.01]
>>> map(lambda x,y:x*y, salary, bonus)
[0.27999999999999997, 0.080000000000000002, 1.04, 0.057999999999999996]
>>> map(lambda x,y:x*y + x, salary, bonus)
[1.68, 0.88000000000000001, 3.64, 5.858]
```

PYTHON

Filter

- Takes a boolean function and applies it to every item in the list

```
>>> agelist=[12,23,78,95,22,36,71,22,20]
>>> filter(lambda x:18<x<50, agelist)
[23, 22, 36, 22, 20]

>>> allnum =list(range(10))
>>> filter(lambda x:x%2==0, allnum)
[0, 2, 4, 6, 8]
```

Reduce

- Takes the first two numbers
- Applies a reduction
- Takes the next number

```
>>> a = list(range(10))
>>> reduce(lambda x,y:x+y, a)
45
```