WHAT WE HAVE LEARNED

How to handle sequences - lists, strings, tuples
TODAY’S CONTENTS

More about Functions
We have learnt about parameters and function arguments:

```python
def create_sun(radius, color):
    sun = Circle(radius)
    sun.setFillColor(color)
    sun.setBorderColor(color)
    sun.moveTo(100, 100)
    return sun

sun = create_sun(30, "yellow")
```

Arguments are mapped to parameters one-by-one, left-to-right.
We can provide default parameters:
```python
def create_sun(radius = 30, color = "yellow"):
    ...
```

Now we can call it like this:
```python
sun = create_sun()
star = create_sun(2)
moon = create_sun(28, "silver")
```

But not like this:
```python
moon = create_sun("silver")
```
Default parameters have to locate after normal parameters

```python
def avg(data, start = 0, end = None):
    if not end:
        end = len(data)
    return sum(data[start:end])/float(end-start)
```

```python
>>> d = [1, 2, 3, 4, 5]
>>> avg(d)
3.0
>>> avg(d, 2)
4.0
>>> avg(d, 1, 4)
3.0
```
The default values are evaluated at the point of function definition in defining scope:

```python
i = 5
def f(arg=i):
    print arg

i=6
f()
```

What is the result?
NORMAL AND DEFAULT PARAMETERS

The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```python
def f(a, L=0):
    L += a
    return L
print f(1)
print f(2)
print f(3)
```

```python
def f(a, L=[]):
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

```python
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

What are the results?
NAMED PARAMETERS (KEYWORD ARGUMENT)

We can include the name of the parameter in the function call to make the code clearer. The order of arguments does not matter.

```python
def avg(data, start = 0, end = None):
    if not end:
        end = len(data)
    return sum(data[start:end])/float(end-start)
```

```python
>>> avg(d, end=3)
2.0
>>> avg(data=d, end=3)
2.0
>>> avg(end=3, data=d)
2.0
>>> avg(end=3, d)
SyntaxError: non-keyword arg after keyword arg
```
def subtract(a, b):
    return a-b
print subtract(5, 2)
print subtract( b = 5, a = 2)
# print subtract( a = 5, 2)
def add(a, b = 3):
    return a+b
print add(5, 2)
print add(5)
# print add(b=3,5)
# print add(b=3,5)
print add(a=3,b=3)
PASS BY REFERENCE VS VALUE

All parameters (arguments) in the Python are passed by reference

def changeme(mylist):
    #This changes a passed list into this function
    mylist.append([1,2,3,4]);
    print "Values inside the function: ", mylist
    return
    # Now you can call changeme function

mylist = [10, 20, 30]
changeme(mylist);
print "values outside the function: ", mylist

What are the results?
What's the result of this code?

```python
def changeme(mylist):
    # This would assign new reference in mylist
    mylist = [1, 2, 3, 4];
    print "Values inside the function: ", mylist
    return
    # Now you can call changeme function

mylist = [10, 20, 30]
changeme(mylist);
print "values outside the function:", mylist
```
You may need to process a function for more arguments than you specified while defining the function.

These arguments are called **variable-length** arguments and are not named in the function definition, unlike required and default arguments.

The general syntax for a function with non-keyword variable arguments:

```python
def funcName([formal_args,] *var_args_tuple):
    #function_body
    ...
```

An **asterisk** (*) is placed before the variable name that will hold the values of all non-keyword variable arguments.
def printinfo( arg1, *vartuple):
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;

printinfo( 10 );
printinfo( 70, 60, 50 );

What are the results?
**ANONYMOUS FUNCTIONS**

**lambda** keyword is used to create small anonymous function

- No return statement. Instead, lambda includes expression that is returned
- lambda is an expression, not a statement
- can access variables that locates in its own local namespace only

**Syntax**

- `lambda[arg[,arg2, …argn]]: expression`

**Example**

```python
def make_incrementor(n):
    return lambda x: x+n
f = make_incrementor(42)
print f(0)
print f(0)
```
ANONYMOUS FUNCTIONS

1)  
L = [ lambda x: x**2, lambda x:x**3, lambda x:x**4 ]

    for f in L :
        print (f(3))

2)  
min = (lambda x, y: x if x<y else y)
print min(101*98, 102*99)
BUILTIN FUNCTIONS FOR LIST USING LAMBDA

```python
>>> data = [1, 10, 9, 23, 17, 24, 7, 12, 30]
>>> print filter(lambda x: x%3 == 0, data)
[9, 24, 12, 30]
>>> print map(lambda x: x*2 +10, data)
[12, 30, 28, 56, 44, 58, 24, 34, 70]
>>> print reduce(lambda x, y: x+y, data)
133
```
BUILT-IN FUNCTIONS FOR LIST

Compute prime numbers in Python

```python
import math

def primes(n):
    nums=range(2,n)
    sqrtsn=math.sqrt(n)
    i=0
    while((nums[i] <= sqrtsn) and (i < len(nums))):
        nums=filter(lambda x: (x==nums[i] or x%nums[i] !=0), nums)
        print nums
        i += 1
    return nums

primes(30)
```