Functions in Python

Functions without arguments are straighforward. You can use the `def` keyword to define a function:

In [1]:
```
def myfunction():
    print 'Called myfunction'
myfunction()
```

```
Called myfunction
```

We can also define functions with arguments:

In [2]:
```
def myfunction2(a, b):
    print 'Called myfunction2(%r,%r)' % (a,b)
myfunction2('avalue', 'bvalue')
```

```
Called myfunction2('avalue','bvalue')
```

When calling a function with arguments, you may also pass the arguments *by name* rather than *positionally.* This is often useful when there are a large number of arguments and you don't want to remember the order in which they appear.

In [3]:
```
myfunction2(b='bvalue first', a='avalue second')
```

```
Called myfunction2('avalue second','bvalue first')
```

`lambda` functions

Python provides a special form of defining functions that consist of nothing more than a single expression using the `lambda` keyword:

In [4]:
```
lambda_adder = lambda a,b: a+b
lambda_adder(1, 2)
```

Out[4]: 3

Note the fact that `lambda` returns the function object itself. This form is often used when a function needs to be passed as an argument to another function, as in a callback. Also note the fact that the `return` statement is implicit.

The equivalent function defined with `def` would be the following

In [5]:
```
def lambda_adder_equiv(a,b):
    return a+b
lambda_adder_equiv(1,2)
```

Out[5]: 3

Default arguments

You can define a function with default argument values. If a value is not passed for an argument with a default value, the default will be used instead:

In [6]:
```
def myfunction3(a, b='default value'):
    print 'Called myfunction3(%r,%r)' % (a,b)
myfunction3('avalue')
```

```
Called myfunction3('avalue','default value')
```

You can, of course, override the default:

In [7]:
```
myfunction3('avalue', 'bvalue')
```

```
Called myfunction3('avalue','bvalue')
```

Variable arguments

We can define a function that takes any number of arguments using the `*args` syntax. The value of the `args` parameter below is any *positional* arguments that remain after accounting for other arguments:

In [8]:
```
def va_adder(prompt, *args):
    print 'Called va_adder(%r, *%r)' % (prompt, args)
    return sum(args)
va_adder('ThePrompt>', 1,2,3)
```

```
Called va_adder('ThePrompt>', *(1, 2, 3))
```

Out[8]:
```
6
```

Likewise, we can call a function with a `tuple` of arguments using a similar syntax:

In [9]:
```
def normal_function(a,b):
    print 'Called normal_function(%r, %r)' % (a,b)
argument_tuple = ('avalue', 'bvalue')
normal_function(*argument_tuple)
```

```
Called normal_function('avalue', 'bvalue')
```

If you want to define a function with variable *keyword* arguments, you can do that as well with the `**kwargs` syntax. In this case, the keyword arguments are passed as a `dict`:

In [10]:
```
def kw_function(**kwargs):
    print kwargs
kw_function(a='avalue', b='bvalue')
```

```
{'a': 'avalue', 'b': 'bvalue'}
```

Of course, we can also pass a dictionary as the keyword arguments of a function:

In [11]:
```
arguments = { 'a': 'avalue', 'b': 'bvalue' }
normal_function(**arguments)
```

```
Called normal_function('avalue', 'bvalue')
```

Exercise

Write a function with the signature `def log(format, *args, **kwargs):` which prints a line, formatted according to the format string. Some sample results are below:

```
>>> log('The pair is (%r,%r)', 1, 2)
The pair is (1,2)
>>> log('The value of a is %(a)r', a='foo')
```

```
The value of a is 'foo'
```