Python CheatSheet (For Linux)
Beginning Python: From Novice to Professional
Chapter 6(Abstraction)

Function

Creating functions is central to structured programming. So how do you define a function? You do this with the **def** (or "function definition")statement:

```
def hello(name):
return 'Hello, ' + name + '!'
```

Documenting Functions

If you want to document your functions so that you're certain that others will understand them later on, you can add comments (beginning with the hash sign, #). Another way of writing comments is simply to write strings by themselves.

```
def square(x):
'Calculates the square of the number x.'
return x*x
```

The doc string may be accessed like this:

```
>>> square.__doc__
'Calculates the square of the number x.'
```

Mutable data structure such as a list:

```
>>> def change(n):
        n[0] = 'Mr. Gumby'
>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> change(names)
>>> names
['Mr. Gumby', 'Mrs. Thing']
```

Modify parameters

Using a function to change a data structure (such as a list or a dictionary) can be a good way of introducing abstraction into your program. The point of abstraction is to hide all the gory details of the updates, and you can do that with functions. Let's first make a function to initialize a data structure: (*** Bellow example for better understanding, you can skip if you want***)

```
def init(data):
    data['first'] = {}
    data['middle'] = {}
    data['last'] = {}
```

In the preceding code, I've simply moved the initialization statements inside a function.
You can use it like this:

```
>>> storage = {}
>>> init(storage)
>>> storage
{'middle': {}, 'last': {}, 'first': {}}
```

Before writing a function for storing names, let's write one for getting them:

```
def lookup(data, label, name):
    return data[label].get(name)
```

With lookup, you can take a label (such as 'middle') and a name (such as 'Lie') and get a list of full names returned. In other words, assuming my name was stored, you could do this:

```
>>> lookup(storage, 'middle', 'Lie')
['Magnus Lie Hetland']
```

It's important to notice that the list that is returned is the same list that is stored in the data structure. So if you change the list, the change also affects the data structure. (This is not the case if no people are found; then you simply return None.) Now it's time to write the function that stores a name in your structure (don't worry if it doesn't make sense to you immediately):

```
def store(data, full_name):
        names = full_name.split()
        if len(names) == 2: names.insert(1, '')
                labels = 'first', 'middle', 'last'
        for label, name in zip(labels, names):
                people = lookup(data, label, name)
                if people:
                        people.append(full_name)
                else:
                        data[label][name] = [full_name]
```

The store function performs the following steps:
1. You enter the function with the parameters data and full_name set to some values that you receive from the outside world.
2. You make yourself a list called names by splitting full_name.
3. If the length of names is 2 (you have only a first and a last name), you insert an empty string as a middle name.
4. You store the strings 'first', 'middle', and 'last' as a tuple in labels. (You could certainly use a list here; it's just convenient to drop the brackets.)
5. You use the zip function to combine the labels and names so they line up properly, and for each pair (label, name), you do the following:
   • Fetch the list belonging to the given label and name.
   • Append full_name to that list, or insert a new list if needed.

Let's try it out:

```
>>> MyNames = {}
>>> init(MyNames)
>>> store(MyNames, 'Magnus Lie Hetland')
>>> lookup(MyNames, 'middle', 'Lie')
['Magnus Lie Hetland']
```

It seems to work. Let's try some more:

```
>>> store(MyNames, 'Robin Hood')
>>> store(MyNames, 'Robin Locksley')
```

Consideration of immutable parameters :

In Python, it's not directly possible; you can modify only the parameter objects themselves. But what if you have an immutable parameter, such as a number? Sorry, but it can't be done. What you should do is return all the values you need from your function (as a tuple, if there is more than one).

Keyword Parameters and Defaults

The parameters we've been using until now are called positional parameters because their positions are important—more important than their names, in fact.

```
def hello_1(greeting, name):
        print '%s, %s!' % (greeting, name)
```

To make things easier, you can supply the name of your parameter:

```
>>> hello_1(greeting='Hello', name='world')
Hello, world!
```

The parameters that are supplied with a name like this are called keyword parameters. you can give the parameters in the function default values:

```
def hello_3(greeting='Hello', name='world'):
        print '%s, %s!' % (greeting, name)
```

The star in front of the parameter puts all the values into the same tuple.

```
def print_params(*params):
        print params
```

Reversing the Process

using the ** operator. Assuming that you have defined hello_3 as before, you can do the following:

```
>>> params = {'name': 'Sir Robin', 'greeting': 'Well met'}
>>> hello_3(**params)
Well met, Sir Robin!
```

stars are really useful only if you use them either when defining a function (to allow a varying number of arguments) or when calling a function (to "splice in" a dictionary or a sequence).

Scoping

There is a built-in function called vars, which returns this dictionary.

```
>>>x = 1
>>>scope = vars()
>>>scope['x']
1
>>>scope['x'] += 1
>>>x
2
```

In general, you should not modify the dictionary returned by vars because, according to the official Python documentation, the result is undefined. In other words, you might not get the result you're after.

This sort of "invisible dictionary" is called a namespace or scope. In addition to the global scope, each function call creates a new one.

The parameters work just like local variables, so there is no problem in having a parameter with the same name as a global variable

Recursion

A useful recursive function usually consists of the following parts:

• A base case (for the smallest possible problem) when the function returns a value directly.

• A recursive case, which contains one or more recursive calls on smaller parts of the problem.

Example:

```
def factorial(n):
        if n == 1:
                return 1
        else:
                return n * factorial(n-1)
```

New Functions in This Chapter

Function	Description
`map(func, seq [, seq, ...])`	Applies the function to all the elements in the sequences
`filter(func, seq)`	Returns a list of those elements for which the function is true
`reduce(func, seq [, initial])`	Equivalent to `func(func(func(seq[0], seq[1]), seq[2]), ...)`
`sum(seq)`	Returns the sum of all the elements of `seq`
`apply(func[, args[, kwargs]])`	Calls the function, optionally supplying argument