Bioinformatics Programming Class 4

September 13, 2011

1 Goal

Last class was about the type `list` and about repetition using the `for` statement. In this class, we will talk about how to organize your code into functions and your functions into modules. We will also speak about another statement for repetition: the `while` statement.

2 Practical

Today there are three exercises: one in each of the three subsequent subsections. The last exercise is optional, but is fun to do nonetheless.

2.1 Functions

We will restructure the calculator that you built in Class 2 into functions. Start by opening up your favorite text editor and creating a new file called `calc.py`. Maybe you've noticed already: every Python script that you wrote so far starts with:

```
#!/usr/bin/python
```

This is the so called *shebang*, which tells your Unix shell that the whole shebang that follows is to be interpreted using Python. So let that be the first line of your `calc.py` script.

Before we continue with the script, first a small intermezzo about functions. In a programming language a *function* is comparable to a mathematical function/map: it takes several arguments as input and returns a value. The biggest difference, however, is that there *may* be side-effects, i.e. subsequent evaluations of a function with exactly the same arguments do not necessarily produce the same return values.

Let's start by defining our very first function `add`, which takes two arguments `a` and `b` as input and returns `a+b`. In code this looks like:

```
def add(a, b):
  return a+b
```

So `def add(a,b)` tells Python: look, a function with the name `add` taking two arguments named `a` and `b` will follow after the colon. Just like with the if and for statements you need to indent the body of a function. In the body you can have all kinds of statements and you can also use `print` to output stuff to the terminal. The result of a function can be returned using the statement `return`.

It's time to experiment a little bit. Fire up a Python shell and enter the following

```
>>> def add(a,b):
...     return a+b
...
```

(Don't forget to press Enter again after the last line!)

Q1. What happens when you evaluate `add(5,4)`?

Q2. What about `add("Hello","World")`?

Q3. Explain why you get an error when doing `add(3,"text")`?

The result of a function can be saved in a new variable. Try the following

```
>>> c = add(5,4)
>>> d = add("Hello","World")
>>> print c
>>> print d
>>> type(c)
>>> type(d)
```

Q4. Why do `type(c)` and `type(d)` differ?

Let's define a new function:

```
>>> def void(a):
...   print "Lalala"
...
```

Try evaluating `void(5)` and `void("test")`. As you can see the function `void` does nothing but printing an annoying string. In fact, if you look more carefully you can see that there is no `return` statement in the body of the function. That's strange, isn't it? Let's experiment some more and try to store the result in a variable.

```
>>> e = void(5)
>>> type(e)
>>> print e
```

So we just encountered yet another type: the `NoneType` whose only value is `None`.

Q5. Try evaluating `None+None`. You should get an error. Can you figure out which operators are allowed on `None`? Hint: try operators that evaluate to `bool`.

Before moving on, let's look at the scope of variables defined inside and outside of functions. The *scope* of a variable is just a fancy name of the exact region in a program of where a particular variable can be accessed. Let's define a new function `test`:

```
>>> def test(a,b):
...   print "a =", a
...   print "b =", b
...   z = a + b
...   print "z =", z
...
```

So in the above function a new variable `z` is declared. Let's try to access it after calling the function:

```
>>> test(5,4)
>>> print z
```

You should get an error, as the scope of variable `z` is *only* the body of the function `test`. In other words, `z` only exists inside of `test`. Now let's define `z` outside of `test` and see whether it will hold the value of the variable `z` local to `test`.

```
>>> z = 0
>>> test(5,4)
>>> print z
```

Q5. Why isn't `z==9`?

Alright, let's go back to our `calc.py` script. In there we will define the functions `add(a,b)`, `sub(a,b)`, `mult(a,b)`, `fdiv(a,b)` (we'll only be considering floating point division).

Your script should look something like this—but do feel free to experiment:

```
#!/usr/bin/python
import sys

def add(a,b):
  return a+b

def sub(a,b):
  return a+b

def mult(a,b):
  return a*b

def fdiv(a,b):
  return a/b
```

The only thing left to do is to handle input from the command line. Since you've already done this several times, the code for that is copied here below.

```
def comp(expr):
  op = expr[0]
  if op == "abs":
    a = float(expr[1])
    return abs(a)
  else:
    a = float(expr[1])
    b = float(expr[2])
    if op == "add":
      return add(a,b)
    elif op == "sub":
      return sub(a,b)
    elif op == "fdiv":
      return fdiv(a,b)
    elif op == "mult":
      return mult(a,b)
    else:
      return "Invalid operator!"

print comp(sys.argv[1:])
```

So let me explain the above code. You can see that there is a function `comp` which takes a list as an argument. In fact, you can see that we invoke `comp` with `sys.argv[1:]` where the latter is a slice of `sys.argv` excluding the first item.

2.1.1 Exercise

1. Extend `calc.py`[1] by implementing the following two new operations:

   - `"fact"` using a new function

   $$\texttt{fact(n)} = n! = \prod_{i=1}^{n} i = 1 \times \ldots \times (n-1) \times n;$$

   - `"sum"` using a new function

   $$\texttt{sum(k, n)} = \sum_{i=k}^{n} i = k + (k+1) + \ldots + n.$$

   Don't forget that in both operations `n` and `k` are integers!

   Hint: make use of the code that you developed in the last class.

[1]http://few.vu.nl/~mer500/calc.py

2.2 Modules

Similar functions can be grouped together in a module. A *module* simply corresponds to a single Python file. In the following we are going to turn `calc.py` into a module and write an interactive calculator.

Create a new file in your favorite text editor called `icalc.py` in the same directory as `calc.py`. Make sure that `icalc.py` looks as follows:

```
#!/usr/bin/python
import calc
```

Now try the following in your Linux shell (make sure that `icalc.py` is executable by using chmod):

```
% ./icalc.py add 4 5
```

So basically the `import calc` statement simply includes the contents of `calc.py` into `icalc.py`. In order to have `calc.py` act more as a module, we need to change it a little: replace the last line `print comp(sys.argv[1:])` with

```
if __name__ == "__main__":
  print comp(sys.argv[1:])
```

Now nothing happens when executing '`./icalc.py add 4 5`' in the shell, which is good! Try executing '`./calc.py add 4 5`'—that is `calc.py` without the 'i'. You can see that `calc.py` is still working the same as it used to.

Time for another intermezzo. Start the Python interpreter again and type the following—after the first statement, you'll be asked for input; just type in some random text

```
>>> a = raw_input("Give me input: ")
>>> print a
>>> type a
```

Q6. Repeat the previous but this time type in a number as input. What is the type of `a`?

Now you know how to get user input! There is one more thing that you need to know, and that you will be using quite a lot in the rest of your Bioinformatics career: the `split` function. Do the following in the interpreter:

```
>>> str = "bananas apples peaches oranges"
>>> basket = str.split()
>>> print basket
```

So you can get a list of substrings out of a bigger string using split. You can specify the so called *delimiter* as an optional argument; try this:

```
>>> str = "bananas,apples,peaches,oranges"
>>> basket = str.split(",")
>>> print basket
```

Q7. How would you split `"bananas, apples, peaches, oranges"`—notice the space after the commas.

Q8. What's the default delimiter?

Let's go back to our script. We now have all the necessary ingredients for our interactive calculator:

```
#!/usr/bin/python
import calc

while True:
  str = raw_input("? ")
  expr = str.split()

  if expr[0] == "stop":
    exit()
  else:
    print calc.comp(expr)
```

Try executing the script:

```
% ./icalc.py
? add 3 3
? sub 3 3
? abs 3
? fact 10
? magic 2 2
? stop
```

You can see that we make use of a `while` statement; this is the topic of the next subsection.

2.2.1 Exercise

2. Later in this course when you hand in code, you have to make sure that it is robust to faulty user input. Your code should not crash, but instead display a proper error message. So the following is not how a proper program should behave—you can see that it crashed because it was expecting an additional argument.

   ```
   % ./icalc.py
   ? crash
   Traceback (most recent call last):
     File "./icalc.py", line 11, in <module>
   ```

```
    print calc.comp(expr)
  File "/Users/melkebir/Documents/Onderwijs/FoB/python/calc.py", line 22, in comp
    a = float(expr[1])
IndexError: list index out of range
```

This assignment is about adding proper error handling. Instead of crashing, the calculator should output an error message on the same input and just continue:

```
% ./icalc.py
? crash
Unsupported operation!
? add 3 3
6
```

Hint: your error handling should be done in the script `calc.py`.

2.3 While loop

We have just seen a while loop in the previous subsection. Similarly to a for statement, the while statement is used for repetition. The main difference, however, is that the while statement is not typically used to iterate over a list. Instead it is used to repeat a sequence of statements *while* a specific condition holds.

Try the following in the Python shell:

```
>>> while True:
...   print "ad infinitum"
...
```

You can press 'CTRL+C' to abort the loop. The previous loop never stops, because `True` is always true. Let's do something more involved, and for instance try to continuously divide a number by two until it becomes 1.

```
>>> number = 64
>>> while number > 1:
...   number = number / 2
...   print number
...
```

In the output you can see that the above while loop terminated as soon as `number` became 1.

Q9. Why is nothing printed when executing the following loop?

```
>>> while 0==1:
...   print "ad infinitum?"
...
```

2.3.1 Exercise

1. This exercise is about a guessing game. Download `thegame.py` from `http://few.vu.nl/~mer500/thegame.py`. Your task to write a script `solve.py` that guesses the correct number:

   ```
   #!/usr/bin/python
   import thegame

   # This initializes a new guessing game,
   # where the number to be guessed is between 1 and 100
   game = thegame.TheGame(100)

   # res == -1 if the number to guess is smaller
   # res == 1 if the number to guess is bigger
   # res == 0 if you won
   res = game.guess(50)
   ```

   Hint: you can win the game by trying out all numbers between 1 and 100. But you can win faster by doing a binary search (google for it!).