

Functions and Modules

MBV-INFx410

Fall 2015

Modularization

- Programs can get big
- Risk of doing the same thing many times
- Functions and modules encourage
 - re-usability
 - readability
 - helps with maintenance

Functions

- Functions found builtin to Python:
 - open(filename, mode)
 - str(number)
- Takes values as parameters, executes code on them, returns results
- Most common way to modularize a program

Functions – how to define

- How to define a function:
Note the :
Part of the
definition!

```
def FunctionName(param1, param2, ...):  
    function code...  
    return data
```
- keyword: def – says this is a function
- functions need names - FunctionName
- parameters are optional, but common
- Function code does something
- keyword return results: return
- return is optional

Function example

```
>>> def hello(name):
...     results = "Hello World to " + name + "!"
...     return results
...
>>> hello()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hello() takes exactly 1 argument (0 given)
>>> hello("Marie")
'Hello World to Marie!'
>>> hello("John")
'Hello World to John!'
>>>
```

Define function using def keyword

Specify what function should do, and what it should return

Running w/o. required parameters: Python complains

Running with req. parameters -> ok

Defining a function in a script

```
def hello(name):
    # This function takes in a name and returns
    # a nice greeting
    results = "Hello World to " + name + "!"
    return results

firstname = "Marie"
functionresult = hello(firstname)
print functionresult
```

Defining the function

Using the function, w. "Marie" as parameter

- Note: difference in indentation between function definition and the rest
- The non-indented code will be run, and will *call* the function, and get the results

Defining a function in a script

```
def hello(name):
    # This function takes in a name and
    # returns a nice greeting
    results = "Hello World to " + name + "!"
    return results

firstname = "Marie"
functionresult = hello(firstname)
print functionresult
```

Define function using def keyword

Using the function, w. "Marie" as parameter

Function scope

- Variables defined inside a function can only be seen there!
- Access the value of variables defined inside of function: return variable

Scope example

```
>>> def test(x):
...     z = 10
...     return "z is " + str(z) + ", x is " + str(x)
...
>>> print test(5)
'z is 10, x is 5'
>>> z = 20
>>> print test(5)
'z is 10, x is 5'
>>> print z
20
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Defining the function,
w. param x, defining
the variable z inside

Testing with x = 5

z defined inside the function,
overrides what is outside

z defined at this level, thus
visible, x defined inside of
function, thus out of scope

Parameters

- Functions can take parameters – not mandatory
- Parameters are positional: follow the order in which they are given

Parameter example

```
>>> def test(string, number):
...     print string.split()
...     print number*number
...
>>> print test("George Adams", 3)
['George', 'Adams']
9
>>> print test("George Adams", "George Adams")
['George', 'Adams']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in test
TypeError: can't multiply sequence by non-int of type 'str'
>>> print test(3, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in test
AttributeError: 'int' object has no attribute 'split'
>>>
```

Defining the function
w. two parameters, one
string, one a number

Testing with good
parameters

First parameter is ok,
but strings cannot be
multiplied

First parameter not ok,
numbers have no split
method. Function fails
before reaching valid
code

Blast parsing script

```
import sys
fh = open(sys.argv[1], "r")
lines = fh.readlines()
fh.close()

fo = open(sys.argv[2], "w")

for line in lines:
    text = line.replace("\n", "")
    fields = text.split()
    name = fields[1]
    start = int(fields[8])
    stop = int(fields[9])
    difference = 0
    if stop < start:
        difference = start - stop
    else:
        difference = stop - start

    outstring = name + "\t" + str(difference) + "\n"
    fo.write(outstring)

fo.close()
```

Read in input file

Open output file for writing.

Get the name and the start and
stop position

Creating a variable to keep the
difference, then calculating it

Creating output string, and writing
it. Notice we added a newline!

Closing the output file

parse_blast_out_function.py

- Create function that looks like this:
def calculate_distance(blastline):
 - create output string containing name and difference, finish with newline
 - return output string
- For loop that, per line in input,
 - Calls calculate_distance
 - Prints output to file
- Template: parse_blast_out_function.py

Adding functions to blast parse script

```
def calculate_difference(blastline):  
    text = blastline.replace("\n","")  
    fields = text.split()  
    name = fields[1]  
    start = int(fields[8])  
    stop = int(fields[9])  
    difference = 0  
    if stop < start:  
        difference = start - stop  
    else:  
        difference = stop - start  
    outstring = name + "\t" + str(difference) + "\n"  
    return outstring
```

Defining the function that does the processing of the blast line, and returns the results

This part of the code remains the same

Return the results to where the function was called

```
# rest of script comes here
```

Parse script cont.

```
# and here the script continues
```

```
# use blastout2.txt as input  
fh = open(sys.argv[1], "r")  
lines = fh.readlines()  
fh.close()
```

This part of the code remains the same

```
# open outfile  
fo = open(sys.argv[2], "w")
```

```
for line in lines:
```

```
    outstring = calculate_difference(line)  
    fo.write(outstring)  
fo.close()
```

Using the function – much easier to follow the logic of what is being done due to less code to read.

Modules

- A module is a file with functions, constants and other code in it
- Module name = filename without .py
- Can be used inside another program
- Needs to be import-ed into program
- Lots of builtin modules: sys, os, os.path....
- Can also create your own

Transforming blast script into module

- May want to use calculate_difference in other scripts
- Can create a module that contains this and possibly other functions too
- Format of the module (script) file:
 - One or more functions, followed by
 - if `__name__ == "__main__"`:
 - Any code that you want to run when you run the script on the command line goes here
- Script run directly on command line: `__name__` will be `== "__main__"`, otherwise not

Blast parse module

```
def calculate_difference(blastline):
    text = blastline.replace("\n", "")
    fields = text.split()
    name = fields[1]
    start = int(fields[8])
    stop = int(fields[9])
    difference = 0
    if stop < start:
        difference = start - stop
    else:
        difference = stop - start
    outstring = name + "\t" + str(difference) + "\n"
    return outstring

if __name__ == "__main__":
    # creating fake input line here
    input = "isotig13419\tcontig698252\t99.79\t472\t1\t0\t1538\t2009\t1187\t716\t0.0\t928\n"
    output = calculate_difference(input)
    print output
```

This part of the code remains the same

If statement triggered if run directly Here: tests function

Save as parse_blast_out_module.py

Using a module

- One of two import statements:
 - 1: import modulename
 - 2: from module import function/constant
- If method 1, inside of script do:
 - modulename.function(arguments)
- If method 2, inside of script do:
 - function(arguments) – module name not needed
 - beware of function name collision

Using calculate_difference in other scripts

```
import sys
from parse_blast_out_module import calculate_difference

fh = open(sys.argv[1], "r")
lines = fh.readlines()
fh.close()

fh = open("results_file.txt", "w")

for line in lines:
    outstring = calculate_difference(line)
    fh.write(outstring)
fh.close()
```

Import statement makes function available

Can use it as if it was defined inside of the current script file

TranslateProteinFunctions.py

- TranslateProteinReadTable.py, readWriteFasta as starting points, get code from there
- Open TranslateProteinFunctions.py file, create functions
- Create following functions:
 - def read_fasta(fastafile):
 - Return list containing header and sequence
 - def read_translationtable(translationtablefile)
 - Return dictionary with codon:aminoacid pairs
 - def translate(sequence, translationtable)
 - Return protein string, translated from DNA sequenc using the translation table
 - def pretty_print(header, sequence, outfile)
 - Print header, and protein sequence in chunks of 60 bp to outfile

Create module

- Copy TranslateProteinFunctions.py to TranslateProteinModule.py
- Add above previously unindented code
 - if `__name__ == "__main__"`:
 - #previous unindented code goes here
- How to use in different script
 - import TranslateProteinModule
- Can now do:
 - fastalist = dnaTranslateProteinModule.read_fasta(filename)
- Code is not present in this file!

formatString.py

- Input: wrongly formatted fasta file (lines are too long)
- Output: pretty printed file

```
import TranslateProteinModule
import sys

inputfile = sys.argv[1]
output = sys.argv[2]

fastalist = TranslateProteinModule.read_fasta(inputfile)
header = fastalist[0]
sequence = fastalist[1]

TranslateProteinModule.pretty_print(header, sequence, output)
```