Python 10

Functions

Functions consist of several lines of code that do something useful and that you want to run more than once. There are built-in functions in python. You can also write your own. You also give your function a name so you can refer to it in your code. This avoids copying and pasting the same code to many places in your script and makes your code easier to read.

Let's see some examples.

Python has built-in functions

```
>>> print('Hello world!')
Hello world!
>>> len('AGGCT')
5
```

You can define your own functions with `def` Let's write a function that calculates the GC content. Let's define this as the fraction of nucleotides in a DNA sequence that are G or C. It can vary from 0 to 1.

First we can look at the code that makes the calculation, then we can convert those lines of code into a function.

Code to find GC content:

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCT'
c_count = dna.count('C')  # count is a string method
g_count = dna.count('G')
dna_len = len(dna) # len is a function
gc_content = (c_count + g_count) / dna_len # fraction from 0 to 1
print(gc_content)
```

Defining a Function that calculates GC Content

We use `def` do define our own function. It is followed by the name of the function (`gc_content`) and parameters it will take in parentheses. If you have more than one parameter, you separate them with commas. A colon is the last character on the `def` line. The parameter variables will be available for your code inside the function to use.

```
def gc_content(dna):   # give our function a name and parameter 'dna'
    c_count = dna.count('C')
    g_count = dna.count('G')
    dna_len = len(dna)
    gc_content = (c_count + g_count) / dna_len
    return gc_content # return the value to the code that called this function
```

> Here is a custom function that you can use like a built in Python function

Using your function to calculate GC content

This is just like any other python function. You write the name of the function with any variables you want to pass to the function in parentheses. In the example below the contents of `dna_string` get passed into `gc_content()`. Inside the function this data is passed to the variable `dna`.

```
dna_string = "GTACCTTGATTTCGTATTCTGAGAGGCTGCT"
print(gc_content(dna_string))
```

This code will print 0.45161290322580644 to the screen. You can save this value in a variable to use later in your code like this

```
dna_gc = gc_content('GTACCTTGATTTCGTATTCTGAGAGGCTGCT')
```

As you can see we can write a nice clear line of python to call this function and because the function has a name that describes what it does it's easy to understand how the code works. Don't give your functions names like this `my_function()`!

How could you convert the GC fraction to % GC. Use `f''`.

```
dna_string = "GTACCTTGATTTCGTATTCTGAGAGGCTGCT"
dna_gc = gc_content(dna_string)
print(f'This sequence is {dna_gc:.2%} GC')
```

Here's the output

```
This sequence is 45.16% GC
```

The details

1. You define a function with `def`. You need to define a function before you can call it.
2. The function must have a name. This name should clearly describe what the function does. Here is our example `gc_content`

3. You can pass variables to functions but you don't have to. In the definition line, you place variables your function needs inside parentheses like this `(dna)`. This variable only exists inside the function.
4. The first line of the function must end with a `:` so the complete function definition line looks like this `def gc_content(dna):`
5. The next lines of code, the function body, need to be indented. This code comprises what the function does.
6. You can return a value as the last line of the function, but this is not required. This line `return gc_content` at the end of our function definition passes the value of gc_content back to the code that called the function in your main script.

Naming Arguments

You should name your argument variables so that they describe the data they contain. The name needs to be consistent within your function.

Keyword Arguments

Arguments can be named and these names can be used when the function is called. This name is called a 'keyword'

```
>>> dna_string = "GTACCTTGATTTCGTATTCTGAGAGGCTGCT"
>>> print(gc_content(dna_string))
0.45161290322580644
>>> print(gc_content(dna=dna_string)  # gc_content is expecting a 'dna' argument
0.45161290322580644
```

> The keyword must be the same as the defined function argument. If a function has multiple arguments, using the keyword allows for calling the function with the arguments in any order.

Default Values for Arguments

As defined above, our function is expecting an argument (`dna`) in the definition. You get an error if you call the function without any parameters.

```
>>> gc_content()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: gc_content() missing 1 required positional argument: 'dna'
```

You can define default values for arguments when you define your function.

```
def gc_content(dna='N'):   # give our function a name and parameter 'dna'
    c_count = dna.count('C')
    g_count = dna.count('G')
    dna_len = len(dna)
    gc_content = (c_count + g_count) / dna_len
    return gc_content # return the value to the code that called this function
```

> If you call the function with no arguments, the default will be used. In this case a default is pretty useless, and the function will return '0' if called without providing a DNA sequence.

Lambda expressions

Lambda expressions can be used to define simple (one-line) functions.

Here is a one line custom function, like the functions we have already talked about:

```
def get_first_codon(dna):
  return dna[0:3]

print(get_first_codon('ATGTTT'))
```

> This will print `ATG`

The format for lambda is like this

```
lambda <the variable you pass into the function> : <the expression that operates on your
variable>
```

Here is the same function written as a lambda

```
get_first_codon = lambda dna : dna[0:3]  # pass data into 'dna' then extract
                                         # the first 3 characters
print(get_first_codon('ATGTTT'))
```

> This also prints `ATG`. lambdas can only contain one line and there is no `return` statement.

List comprehensions can often be used instead of lambdas and may be easier to read. You can read more about `lambda`, particularly in relation to `map` which will perform an operation on a list, but generally a `for` loop is easier to read.

Scope

Almost all python variables are global. This means you can use them everywhere in your code. Remember that python blocks are defined as code at the same level of indentation.

```
#!/usr/bin/env python3
print('Before if block')
x = 100
print('x =',x)
if True:  # this if condition will always be True
  # we want to make sure the block gets executed
  # so we can show you what happens
  print('Inside if block')
  x = 30
  y = 10
  print("x =", x)
  print("y =", y)

print('After if block')
print("x =", x)
print("y =", y)
```

Let's Run it:

```
$ python3 scripts/scope.py
Before if block
x = 100
Inside if block
x = 30
y = 10
After if block
x = 30
y = 10
```

The most important exception to variables being global is that variables that are defined in **functions** are **local** i.e. they only exist inside their function. Inside a function, global variables are visible, but it's better to pass variables to a function as arguments

```
def show_n():
  print(n)
n = 5
show_n()
```

The output is this `5` as you would expect, but the example below is better programming practice. Why? We'll see a little later.

```
def show_n(n):
  print(n)
n = 5
show_n(n)
```

Local Variables

Variables inside functions are local and therefore can only been accessed from within the function block. This applies to arguments as well as variables defined inside a function.

```
#!/usr/bin/end python3

def set_local_x_to_five(x):
  print('Inside def')
  x = 5 # local to function set_local_x_to_five()
  y = 5   # also local
  print("x =",x)
  print("y = ",y)

print('After def')
x = 100 # global x
y = 100 # global
print('x =',x)
print('y =',y)

set_local_x_to_five(500)
print('After function call')
print('x =',x)
print('y =',y)
```

> Here we have added a function `set_local_x_to_five()` with an argument named `x`. This variable exists only within the function where is replaces any variable with the same name outside the `def`. Inside the `def` we also initialize a variable `y` that also replaces any global `y` within the `def`

Let's run it:

```
$ python3 scope_w_function.py
After def
x = 100
y = 100
Inside def
x = 5
y =  5
After function call
x = 100
y = 100
```

> There is a global variable, `x` = 100, but when the function is called, it makes a **new local variable**, also called `x` with value = 5. This variable disappears after the function finishes and we go back to using the global variable `x` = 100. Same for `y`

Global

You can make a local variable global with the statement `global`. Now a variable you use in a function is the same variable as in the rest of the code. It is best not to define any variables as global until you know you need to because you might modify the contents of a variable without meaning to.

Here is an example use of `global`.

```
#!/usr/bin/env python3

def set_global_variable():
  global greeting  # make greeting global
  greeting = "I say hello"


greeting = 'Good morning'
print('Before function call')
print('greeting =',greeting)

# make call to function
set_global_variable()
print('After function call')
print('greeting =',greeting)
```

Let's look at the output

```
$ python3 scripts/scope_global.py
Before function call
greeting = Good morning
After function call
greeting = I say hello
```

> Note that the function has changed the value of the global variable. You might not want to do this.

By creating new local variables inside function definitions, python stops variables with the same name from over-writing each other by mistake.

Modules

Python comes with some core functions and methods. There are many useful modules that you will want to use. `import` is the statement for telling your script you want to use code in a module. As we've already seen with regular expresions, you can bring in code that handles regular expressions with `import re`

Getting information about modules with `pydoc`

How do you find out information about a module? Python has help pages built into the command line, like `man` we met earlier in the unix lecture. Online information may be more up to date. Search at https://docs.python.org/3.9/. But if you don't have internet access, you can always use `pydoc`.
To find out about the `re` module, type `pydoc re` on the command line. The last line in the output tells you where the python module is actually installed.

```
% pydoc re
Help on module re:

NAME
    re - Support for regular expressions (RE).

MODULE REFERENCE
    https://docs.python.org/3.9/library/re

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides regular expression matching operations similar to
    those found in Perl.  It supports both 8-bit and Unicode strings; both
    the pattern and the strings being processed can contain null bytes and
    characters outside the US ASCII range.
```

```
    Regular expressions can contain both special and ordinary characters.
    Most ordinary characters, like "A", "a", or "0", are the simplest
    regular expressions; they simply match themselves.  You can
    concatenate ordinary characters, so last matches the string 'last'.

    The special characters are:
        "."      Matches any character except a newline.
        "^"      Matches the start of the string.
        "$"      Matches the end of the string or just before the newline at
                 the end of the string.
```

Here are some of the most common and useful modules, along with their methods and objects. It's a lightning tour.

os.path

`os.path` has common utilities for working file paths (filenames and directories). A path is either a relative or absolute list of directories (often ending with a filename) that tells you where to find a file or directory.

function	description
os.path.basename(path)	what's the last element of the path? Note `/home/tmp/` returns `''`, rather than `tmp`
os.path.dirname(path)	what's the directory the file is in?
os.path.exists(path)	does the path exist?
os.path.getsize(path)	returns path (file) size in bytes or error
os.path.isfile(path)	does the path point to a file?
os.path.isdir(path)	does the path point to a directory?
os.path.splitext(path)	splits before and after the file extension (e.g. '.txt')

`__file__` is the path to your current python script

os.system

Replaced by subprocess.

subprocess

This is the current module for running command lines from python scripts

```
import subprocess
subprocess.run(["ls","-l"])  # same as running ls -l on the command line
```

more complex than `os.system()`. You need to specify where input and output go. Let's look at this in some more detail.

Capturing output from a shell pipeline

Let's say we want to find all the files that have user amanda (or in the filename)

`ls -l | grep amanda`

We can write the following code to capture the output of the two unix commands in the variable `output`

```
import subprocess
output = subprocess.check_output('ls -l | grep amanda', shell = True)
```

This is better than alternatives with `subprocess.run()`. This is the python equivalent of the unix backtick quoted string `ls -l | grep amanda`.

`output` contains a bytes object (more or less a string of ASCII character encodings)

```
b'-rw-r--r--  1 amanda  staff       161952 Oct  2 18:03 test.subreads.fa\n-rw-r--r--  1
amanda  staff          126 Oct  2 13:23 test.txt\n'
```

You can covert by decoding the bytes object into a string

```
>>>output.decode('utf-8')
'-rw-r--r--  1 amanda  staff       161952 Oct  2 18:03 test.subreads.fa\n-rw-r--r--  1
amanda  staff          126 Oct  2 13:23 test.txt\n'
```

Capturing output the long way (for a single command)

Let's assume that `ls -l` generates some output something like this

```
total 112
-rw-r--r--  1 amanda  staff           69 Jun 14 17:41 data.cfg
-rw-r--r--  1 amanda  staff       161952 Oct  2 18:03 test.subreads.fa
-rw-r--r--  1 amanda  staff          126 Oct  2 13:23 test.txt
```

How do we run `ls -l` in Python and capture the output (stdout)?

```
import subprocess
rtn = subprocess.run(['ls','-l'], stdout=subprocess.PIPE )  # specify you want to capture
STDOUT
bytes = rtn.stdout
stdout = bytes.decode('utf-8')
# something like
lines = stdout.splitlines()
```

`lines` now contains elements from every line of the `ls -l` output, including the header line, which is not a file

```
>>> lines[0]
'total 112'
>>> lines[1]
'-rw-r--r--  1 amanda  staff     69 Jun 14 17:41 data.cfg'
```

Check the exit status of a command

To run a command and check the exit status (really to check the exit status = 0, which means success), use

```
oops = subprocess.check_call(['ls', '-l'])
# or, simpler...
oops = subprocess.check_call('ls -l', shell=True)
```

Run a command that redirects stdout to a file using python subprocess

You can't write `ls -l > listing.txt` to redirect stdout in the subprocess method, so use this instead

```
 tmp_file = 'listing.txt'
 with open(tmp_file,'w') as ofh:
     oops = subprocess.check_call(['ls', '-l'], stdout=ofh )
```

sys

A couple of useful variables for beginners. Many more advanced system parameters and settings that we are not covering here.

function	description
sys.argv	list of command line parameters
sys.path	where Python should look for modules

re

See notes on regular expressions

collections

Better lists etc.

`from collections import deque`

copy

`copy.copy()`

and

`copy.deepcopy()`

Link to more info for more on deep vs shallow copying

math

function	description
math.exp()	e**x
math.log2()	log base 2
math.log10()	log base 10
math.sqrt()	square root
math.sin()	sine
math.pi(), math.e()	constants
etc	

see also numpy

random

Random numbers generated by computers are not truly random, so python calls these pseudo-random.

example	description
random.seed(1)	set starting seed for random sequence to 1 to enable reproducibility
random.randrange(9)	integer between 0 and 8
random.randint(1,5)	integer between 1 and 5
random.random()	float between 0 and 1
random.uniform(1,2)	float between 1 and 2
random.choice(my_genes)	return a random element of the sequence

To get a random index from an element of `list` use `i=random.randrange(len(list))`

statistics

Typical statistical quantities

example	description
statistics.mean([1,2,3,4,5])	mean or average
statistics.median([2,3,4,5])	median = 3.5
statistics.stdev([1,2,3,4,5])	standard deviation of sample (square root of sample variance)
statistics.pstdev([1,2,3,4,5])q	estimate of population standard deviation

glob

Does unix-like wildcard file path expansion.

```
>>> import glob
>>> glob.glob('pdfs/*.pdf')
['pdfs/python1.pdf', 'pdfs/python2.pdf', 'pdfs/python3.pdf', 'pdfs/python4.pdf',
'pdfs/python6.pdf', 'pdfs/python8.pdf', 'pdfs/unix1.pdf', 'pdfs/unix2.pdf']
>>> fasta_files = glob.glob('sequences/*.fa')
>>>
```

argparse

Great (if quite complicated) tool for parsing command line arguments and automatically generating help messages for scripts (very handy!). Here's a simple script that explains a little of what it does.

```
#!/usr/bin/env python3
import argparse
parser = argparse.ArgumentParser(description="A test program that reads in some number of
lines from an input file. The output can be screen or an output file")
# we want the first argument to be the filename
parser.add_argument("file", help="path to input fasta filename")
# second argument will be line number
# default type is string, need to specify if expecting an int
parser.add_argument("lines", type=int, help ="how many lines to print")
# optional outfile argument specified with -o or --out
parser.add_argument("-o","--outfile", help = "optional: supply output filename, otherwise
write to screen", dest = 'out')
args = parser.parse_args()
# arguments appear in args
filename = args.file
lines = args.lines
if args.out:
  print("writing output to", args.out)
```

With this module, -h help comes for free. --outfile type arguments are optional unless you write 'required=True' like this

```
parser.add_argument('-f', "-fasta", required=True, help='Output fasta filename',
dest='outfile')
```

Here's an example python template that uses several different capabilities of the argparse module.

Many more modules that do many things

time, HTML, XML, email, CGI, sockets, audio, GUIs with Tk, debugging, testing, unix utils

Also, non-core: BioPython for bioinformatics, Numpy for mathematics, statistics, pandas for data, scikitlearn for machine learning.

Your own modules

You can also make your own modules. They are just text files containing python. The file name should end with `.py`. You need to put them in the right directory (same directory as your main script works for getting going). Then you can write your own import statement. For example

```
#!/usr/bin/env python3
import sequence_utilities    # import functions in your own file 'sequence_utilities.py'
import sys                   # import built-in python module
```