Fun with functions!

A function is a block of code that performs a single action that can be reused indefinitely. They provide a great way of breaking code into smaller reusable pieces. And they're fun!

Built-in and user-defined functions

We have seen serveral built-in functions now: `print()` , `len()` , `input()` , `type()` , `float()` , etc. Note that the syntax is similar - *function_name(arguments)*. When we call a function, it executes some code using as input arguments passed into the function, such as variables and values, and returns a result. More simply stated, a function takes arguments, executes code, and returns a result. This result is called a *return value*.

For example, the length function `len()` takes as an argument an object, such as a string, some code runs in the background that determines the number of items in the object, for a string this would the number of characters, and a value is returned.

```
In [1]: len('ATGTAC')
```

```
Out[1]: 6
```

See https://docs.python.org/3.4/library/functions.html (https://docs.python.org/3.4/library/functions.html) for a list and description of the ~70 built in functions. Go through the list and explore them on your own.

Python makes it easy to create your own functions. These functions are completely portable and can be used in any Python script essentially just like the built-in functions.

We can define a function by assigning a name to the function and a block of code to execute - this is referred to as *function definition*. The syntax is as follows:

```
In [ ]: def function_name(arguments):
            block of code
```

`def` is a keyword that is used to indicate that this is a function definition. The first line is called the header and the block of code is called the body and must be indented.

Define a function that simply contains a print statement:

```
In [3]: def joke():
            print('What do houses and computers have in common?\n Bugs come in th
```

The empty parentheses indicate that no arguments are required:

Now we can call our new function just like we would a built-in function:

```
In [4]: joke()
```

```
What do houses and computers have in common?
 Bugs come in through open Windows
```

The function prints something but doesn't actually produce a value that is useful downstream. Recall that the built-in functions we have seen produce a useful value when called. The function does produce a value, but that value is `None` :

```
In [5]: value  = joke()
```

```
What do houses and computers have in common?
 Bugs come in through open Windows
```

```
In [6]: print(value)
```

```
None
```

A function that doesn't return a value is called a void function.

Exercise 4a

Define a function called `echo()` that prints whatever argument is passed to it, similar to the `print()` function, or a common unix command. Use the code box below to define the function and then the next code box to call the function.

```
In [7]: def echo(text):
            print(text)
```

```
In [8]: echo('Helooooooo')
```

```
Helooooooo
```

Arguments are passed to variables within the function called parameters. `text` , in the example above, is the stand in for the actual arguments passed by the user.

Returning values

For a function to be useful, it typically should return a value.

Define a function called `concatenator()` that concatenates two DNA or RNA sequences that we provide as arguments:

In [9]:
```
def concatenator(seq1, seq2):
    return seq1 + seq2
```

The `return` statement is used to provide a value from a function. The `return` statement also exits the function so you can only have one `return` statement and you can not include any code after the `return` statement.

Now we can use our new function just like a built-in function:

In [12]:
```
concatenator('ATG', 'TGA')
```

Out[12]:
```
'ATGTGA'
```

Notice that the function we defined specifically requires two arguments, `seq1` and `seq2` (what you call these variables is arbitrary), which are then used in place of the actual seq1 and seq2 in the function definition. Naming of functions follows the same general rules as naming variables.

Define a function called `base_id()` that takes as an argument a nucleic acid sequence and returns dna if it contains Ts and rna if it contains Us:

In [13]:
```
def base_id(seq):
    if ('U' in seq) or ('u' in seq):
        return 'rna'
    elif ('T' in seq) or ('t' in seq):
        return 'dna'
    else:
        return 'unk'
```

We can confirm that `base_id()` is a function using the `type` function:

In [14]:
```
type(base_id)
```

Out[14]:
```
function
```

Call the function:

```
In [17]: base_id('AAA')
Out[17]: 'unk'
```

When we call a function we have to provide the required arguments, in this case a single sequence, which is then assigned to the parameter `seq` that we specified in the function. Just as with built-in function, the argument can be a variable.

Assign a sequence to a variable:

```
In [18]: sequence = 'ATG'
```

Call the `base_id()` function providing the variable as the argument:

```
In [19]: base_id(sequence)
Out[19]: 'dna'
```

If we were to call the function in a script, just as with built-in functions such as `len()`, a result is returned but it is not printed or acted on. We can store the return value as a variable if that is useful or we could modify the function to print the result.

```
In [20]: base = base_id(sequence)
         print(base)

dna
```

Exercise 4b

In the code box below, define a function that returns the product of two numbers.

```
In [21]: def product(n1, n2):
             return n1 * n2
```

Call the function:

```
In [22]: product(7, 5)
Out[22]: 35
```

Setting default argument values

It is often useful to have default values for one or more arguments. Modify the concatenator function to concatenate 1-4 sequences:

```
In [23]: def concatenator(seq1, seq2 = '', seq3 = '', seq4 = ''):
             return seq1 + seq2 + seq3 + seq4
```

The empty values (specified by `''`) asssigned to seq2 - seq4 will be automatically used if the user provides less than 4 arguments:

```
In [27]: concatenator('ATG', 'TTT', 'AAA')

Out[27]: 'ATGTTTAAA'
```

Local vs global variables

Variables assigned within a function cannot be used outside of the function, these are called local variable becuase they can only be used locally. Variable assigned outide of a function are global, meaning they can be used anywhere.

```
In [29]: sequence = 'ATGTGA'
         print(sequence)
         print(seq1)

ATGTGA

-------------------------------------
NameErrorTraceback (most recent call last)
<ipython-input-29-13144164ec9a> in <module>()
      1 sequence = 'ATGTGA'
      2 print(sequence)
----> 3 print(seq1)

NameError: name 'seq1' is not defined
```

Functions within scripts

A function can be defined within a script but the function is not actualy executed unless the function is called. It can be used within the script at any point after it is defined. After the statements within the function are executed, the script will pick up right where it left off when the function was called.

Exercise 4c

Define a function that calculates the sum of the squares of two numbers. Call the function within a script that prompts the user for the two numbers and prints the value returned by the function to the terminal. The `input()` and `print()` statements should not be within the sum of squares function.

Modules

What if we want to define a function that we can use in other scripts or call in the Python interpretor or ipython notebook? This is easily done by placing a function, possibly along with related functions, into a seperate document. A document that contains Python function definitions that can be imported into other scripts is called a module and has the extension `.py` .

Copy the code for the `concatenator()` and `base_id()` functions into a new text editor file and save the file as `seqedit.py` .

Document strings and comments will help make your modules and functions easier to read:

```
"""
The message contained here in triple quotes is called a documenta
tion string (docstring)

It is the first statement in a module and after importing the mod
ule can be accessed with help(module_name)

It should contain a brief description of the module

See http://www.pythonforbeginners.com/basics/python-docstrings fo
r conventions
"""
def function_name(arguments):
    """
    Each function should also have a docstring with a brief descr
iption of what the function does

    It can be accessed with help(module_name.function_name)
    """
    block of code

if __name__ == '__main__':
    """
    This part of the program is used to test code when a module i
s run as a script, as described below
    """
    print(modulename(arguments))
```

Before custom functions can be called within a script or within the Python interpretor, the module that they are contained in has to be imported into the program. Import the `seqedit` module (the file has to be in the same location as this notebook):

In [30]:
```
import seqedit
```

Notice that when we import the function, we leave off the `.py` extension.

In [32]:
```
help(seqedit.base_id)
```

```
Help on function base_id in module seqedit:

base_id(seq)
    Identifies if a sequence is DNA or RNA.
```

To call a specific function within the module, use the syntax `module_name.function_name()` (also called dot notation). Try calling the `base_id()` function:

```
In [35]: seqedit.base_id('AAA')
```

```
Out[35]: 'unk'
```

Any function within the module can be called using the syntax above. Reloading a module, which you might have to do if you have an error in the original module, is overly complicated in Python3. You first have to import the importlib package (a package is a collection of modules) and then use the reload function contained in the package:

```
In [37]: import importlib
         importlib.reload(seqedit)
```

```
Traceback (most recent call last):

  File "//anaconda/lib/python3.6/site-packages/IPython/core/interactiv
eshell.py", line 2963, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-37-ccfd49b4bb64>", line 2, in <module>
    importlib.reload(seqedit)

  File "//anaconda/lib/python3.6/importlib/__init__.py", line 166, in
reload
    _bootstrap._exec(spec, module)

  File "<frozen importlib._bootstrap>", line 618, in _exec

  File "<frozen importlib._bootstrap_external>", line 674, in exec_mod
ule

  File "<frozen importlib._bootstrap_external>", line 781, in get_code

  File "<frozen importlib._bootstrap_external>", line 741, in source_t
o_code

  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames
_removed

  File "/Users/montgomery/Dropbox/DSCI511/seqedit.py", line 10
    return seq1 + seq2 + seq3 + seq4
                                    ^
TabError: inconsistent use of tabs and spaces in indentation
```

When you import a module, a new folder __pycache__ will appear in the directory containing the module. This is bytecode generated by the Python interpretor that is used when a function is a called to save time in executing the code.

Where to find information on available modules

There are numerous standard Python modules that can be imported into a program at any time. See https://docs.python.org/3/py-modindex.html (https://docs.python.org/3/py-modindex.html) for a list and descriptions.

There are also a of number modules freely available that accomplish routine tasks in biology. See for example BioPython - http://biopython.org/wiki/Documentation (http://biopython.org/wiki/Documentation) and http://biopython.org/DIST/docs/api/module-tree.html (http://biopython.org/DIST/docs/api/module-tree.html).

Also see https://pypi.python.org/pypi?%3Aaction=search&term=bioinformatics&submit=search (https://pypi.python.org/pypi?%3Aaction=search&term=bioinformatics&submit=search) for additional Bioinformatics packages.

Exercise 4d

1) Define a function, `abs_val()` that returns the absolute value of a number (there is a built-in function `abs()`, so do not name the function `abs`). This can be done in the code box below.

In [38]:
```
def abs_val(n):
    return (n**2)**(1/2)
```

2) Call the function providing a negative value as an argument.

In [39]:
```
abs_val(-5)
```

Out[39]:
```
5.0
```

3) Create a module called `numbers` containing the `abs_val()` function.

4) Write a script that prompts the user for a number, calls the `abs_val()` function, and prints the absolute value of the number. The script should import the `numbers` module, using `import numbers` so that the `abs_val()` function can be called.

Testing functions

Modules can be imported into the Python interpreter, which provides a useful way of testing functions.

Launch the Python interpreter and import the `numbers` module and test the `abs_val()` function from the previous exercise.

When you execute a script in Python, as the initial file executed it is assigned the name `main` . A module imported into Python is assigned a different name. If you want to make a module that can also act a standalone program, you can use the following syntax to include additional code that is executed only when the module is executed as a script:

In [ ]:
```
def modulename()
    block of code

if __name__ == '__main__':

    block of code
```

The block of code in the `if` statement typically passes arguments to a function in the module and prints the *return value* of the function. For example:

In [ ]:
```
if __name__ == '__main__':

    print(modulename(arguments))
```

By placing it within the `print()` function we can call the function and print the *return value*.

Questions

Q. What is a function?

A.

Q. What are some advantages to defining functions?

A.

Q. What is a module?

A.

Q. What does the following jargon mean: *function definition*, *call a function*, *pass an argument*, *return a value*?

A.

Assignment 2

Assignment 2 is now available. It is due by 10 am on 10/2.

```
In [ ]:
```

Present Slides Themes Help