Programming for Biologists

Learning Objectives

- Review Functions
- Creating and using modules
- Input and Output (I/O)
    - File handling
- List comprehension
- Example function with all the above

Function review

- Function definitions are compound statements

```
def name(parameter-list): # starts with a colon
    body                  # indent 4 spaces
```

```
def validate_base_seq(nt_seq,RNAflag=False):
    '''return True if base seq contains
    only ( acgtACGT and RNAFlag=False )
    or ( acguACGU and RNAflag=True )'''
    seq = nt_seq.upper()
    return len(seq) == (seq.count('A') +
        seq.count('U' if RNAflag else 'T') +
        seq.count('C') + seq.count('G'))
```

Comments vs. Docstring

```
def validate_base_seq(nt_seq,RNAflag=False):
    '''return True if base seq contains
    only ( acgtACGT and RNAFlag=False )
    or ( acguACGU and RNAflag=True )'''
    seq = nt_seq.upper()
    return len(seq) == (seq.count('A') +
        seq.count('U' if RNAflag else 'T') +
        seq.count('C') + seq.count('G'))
```

help function for docstrings

- Comments disappear when python interprets code.
- Docstrings are retained, and therefore have greater utility than comments.
- The help function looks at the docstring of a user-defined function together with its parameter list to generate a help description:

```
help(translate_dna)
```

```
Help on function translate_dna in module __main__:

translate_dna(nt_seq)
    return one-letter AA code if argument
    is a valid sequence string, else notify
```

Multi-line statements

- If an expression contains open parentheses, the interpreter automatically looks for the continuation on the next line.
- You can also use backslashes to indicate line continuation.

```
def validate_base_seq(nt_seq):
    seq = nt_seq.upper()
    return len(seq) == (seq.count('A') + seq.count('T') +
                        seq.count('C') + seq.count('G'))
```

```
def validate_base_seq(nt_seq):
    seq = nt_seq.upper()
    return len(seq) == \
        seq.count('A') + seq.count('T') + \
        seq.count('C') + seq.count('G')
```

Default Function Parameters

- You can define default parameters in a function
- In the parameter list, these must always come **AFTER** parameters that have no default value:

```
def validate_base_seq(nt_seq,RNAflag=False):
    '''return True if base seq contains
    only ( acgtACGT and RNAFlag=False )
    or ( acguACGU and RNAflag=True )'''
    seq = nt_seq.upper()
    return len(seq) == (seq.count('A') +
        seq.count('U' if RNAflag else 'T') +
        seq.count('C') + seq.count('G'))
```

```
validate_base_seq('AUG',True)
```

```
True
```

```
validate_base_seq('AUG') # or ('AUG',False)
```

```
False
```

Reserved Keywords

- Some words cannot be used as user-defined names in programs. These include the following:
  - Value names: True, False, None
  - Operators: and, not, or, in, del, is
  - Definitions: def, class, as, import, from, global, nonlocal, lambda
  - General: assert
  - Conditionals: if, else, elif
  - Flow control: pass, return, yield, while, for, break, continue
  - Exception handling: try, except, finally, with, raise

Scope

- Note that the variables used as function arguments or as local variables do not clash with global variables defined outside of the function.
- Variables inside a function (local) have a limited or nested scope.
- This allows us to choose a variable name that makes sense when writing a function without worrying variable names outside the function (global).

Using Modules

Python offers a large number of optional types, functions, and methods.

- These are defined by **module** files, which live in a library directory in your computer's Python installation.
- You can also define your own modules for use by other programs. These should live in a separate library area within your own workspace.
- The contents of a module is brought into the interpreter's environment using an **import statement**:
- Note: name is just the module's name, i.e. the prefix of the .py filename, with no path and no .py extension.

Modules and Namespaces

Each module has its own namespace, which is isolated from namespaces of other modules and the interpreter's namespace.

- This means that functions with the same name can be defined in multiple namespaces and will not interfere with each other.
- Contents of imported modules can be accessed using the same dot notation used for method calls:

```
import os # python interface to the operating system
os
```

```
<module 'os' from '/home/mkatari/miniconda3/envs/jupyterlab/lib/python3.8/os.py'>
```

```
os.getcwd()
```

```
'/home/mkatari'
```

Selective Module Name Import

Specific names from a module can be imported into the namespace

```
from modulename import name1, name2 ...
```

In this case, the imported name can be used without the dot syntax.

You can rename imported names to avoid conflicts between namespaces:

```
from modulename import actualname as newname ...
```

Selective Module Name Import

You can import ALL names from a module thus:

```
from modulename import *
```

However this is not usually a good idea because you might accidentally be clobbering a name you've defined differently in the second namespace.

In general, avoiding the selective import makes your code clearer because any imported name will be prefixed with the name of its module, and any name without a prefix must be defined in the current namespace.

Example: point mutagenesis

```
from random import randint
def random_mutation(base):
    '''given a specific base, return any of the other three bases'''
    bases='ACGT'
    bases = bases.replace(base, "") # remove the base to replace
    return bases[randint(0,2)]      # randomly select one of the other
```

```
test_seq = 'ACCC'
new_seq=[]
for i in range(len(test_seq)):
    new_seq.append(random_mutation(test_seq[i]))

print("".join(new_seq))
```

```
GATA
```