Syntax and Semantics of Coding in Python
Jeremy Meulemans, Tonya Ward, and Dan Knights

Abstract
Python is a dynamically typed programming language providing a unique blend of power, simplicity and expressiveness that has quickly established itself as a major player in technical fields. The language is built around a very natural syntax, making it an ideal language for both beginners to programming and scientists looking for rapid research implementation. Usage cases from simple file formatting to determining the impact of oil degradation in obtained metagenomic sequence data are facilitated by features of the language. For example, the study of oil breakdown by bacterial communities is likely to involve handling very high-dimensional data, including both the metabolites present in oil and the hundreds of bacterial species often present in environmental communities. The ease of incorporating existing Python modeling packages and the solid frameworks provided by python for scientific computing make this an ideal language for analyzing these types of data.

Keywords: Coding, Introduction, Language, Problem solving, Python

1 Introduction
Python is a rich language with widespread use in education, industry, and research. Python is largely chosen in these fields for its readable syntax, fast development speed, and expressiveness as a general purpose tool for problem solving. From our experience, programs written in languages such as C++ and Java can be written in fewer lines of code, with a higher degree of readability and abstraction, with Python. In comparison to R, Python tends to emerge as a more general use language suited to a wider variety of applications and larger projects, although R does arguably outperform in some areas such as early integration of advanced statistical tools.

For researchers working with hydrocarbons and lipids, a good example of the actual usage of Python is given by Mason et al. [1]. In their analysis of the Deepwater Horizon oil spill’s effect on microbial communities, raw DNA sequencing data from sediment samples were processed by a Python package called QIIME [2] to quality filter the reads. The cleaned data was then loaded into a
package called USEARCH [3], to perform clustering, and then aligned using another Python package called PyNAST [4]. The results of this Python pipeline were used to reveal that samples exhibiting high levels of polycyclic aromatic hydrocarbons had an abundance of genes involved with denitrification pathways.

As Python is a general language, it would be impossible to consider all possible use cases; however, a strong foundation of syntax and functionality of programming in Python should serve as a basis for advanced usages like that of the previous example. To get the most out of the following overview, it is best to read with a mixture of caution and curiosity. For example, when reading the following explanation of conditional statements, it is natural to wonder if the code executed by each conditional statement will be the same if its operands are swapped, or if statement ordering matters, or if if statements are allowable among elif statements. While covering all such cases is difficult, what is beautiful about Python is the rapidity and ease with which code can be written and experimentally tested. As we will explain, one can simply create a file with the different features of the language that are in question, with the use cases of interest, and run it. The output of the interpreter (or the error messages generated if running fails) will serve to quickly refine and answer many questions.

In the following sections, the basics of Python will be established, followed by a larger example of problem solving using Python. This is intended as quick reference to the important features of the language from the perspective of a biological researcher with little to no computer science background and is therefore by no means complete. The material has been tailored to the elements that are most used in practice, but the depth of Python is far greater than this guide. However, the guide should serve as an adequate starting point. Fortunately, Python’s popularity has insured that the Internet is an incredibly rich resource for further information and troubleshooting with the language beyond what can be accomplished or explained by one’s own use and experimentation.

2 Installation

Python is available for all major platforms, and in the case of many Linux distributions and OSX, it comes installed on the system. Installation is usually straightforward, and issues that arise can often be readily troubleshooted through a simple web search. In this guide, the version of Python used is version 2.7.6. The print function changes from print to print() in later versions, which is the only feature that should cause deviations from examples given below if a later version is used.

The command line is where most of the functionality of Python lies, so basic familiarity is a prerequisite. For this guide, this includes
primarily navigating into and out of directories from the command line. Many tutorials exist on this topic, and again a simple web search should suffice for any questions.

Once Python is successfully installed, try typing python into the command line. An interpreter should come up with information about the current version of Python installed on the system. Try typing 2+2 into this interpreter, or print "hello world". To exit, type quit(). This demonstrates some of the basic interaction with the Python interpreter. In the following examples, >>> will make explicit the input of commands typed into this interpreter.

### 3 Types, Structures, and Basic Usage

#### 3.1 Numbers and Importing

Integers and floating point numbers are handled in an intuitive way in Python. The following are all integers:

```
1
13
736205637
```

and the following are all floats:

```
1.0
13.45
736205637.0
```

These values can be stored in variables in a predictable way, with math also behaving as expected:

```
>>> int1 = 3
>>> int2 = 5
>>> float1 = 3.0
>>> float2 = 5.67
>>> floatPi = 3.14159
>>> print int1 + int2 + int2 - int1
10
>>> print float1 * float2
17.01
>>> print int2 ** float1
125.0
>>> diam = int1 + int2
>>> area = floatPi * (diam/2.0)**2.0
>>> print area
50.26544
```

Note that if both operands are integers for division in Python, integer division will be performed, so 5/2 = 2, 5/2.0 = 2.5, and 5.0/2.0 = 2.5.

Although the creation of functions will be explored later in this guide, the usage of functions is relatively straightforward. If the function is built-in to Python or user defined, the function is callable with the `function(argument1, argument2, ...) syntax.`
If the function originated with some module that was imported, as is seen below, the module.function(arg1, arg2, ...) syntax is used.

In the case of mathematics in Python, further functionality for math is implemented in the provided math module. The functionality of this module is imported into the interpreter, thereby making the functions of the module callable. Functionality for third party packages, once installed, is imported in much the same way (see Sect. 4.2):

```python
>>> import math
>>> math.sqrt(9.0)
3.0
>>> math.ceil(5.0/2)
3
>>> math.pow(3, 3) == 3**3
True
>>> help(math)
Help on module math:
```

The last statement shows how to view the full functionality of a module in Python. The help(module) syntax provides a succinct but comprehensive listing of all the functions that are provided by a module, provided the module creator has implemented such a help listing.

### 3.2 Booleans

The previous example also revealed another of the basics types of Python, the Boolean type that can have values True and False. For example:

```python
>>> 3 == 3.0
True
>>> 3 != 3.0
False
>>> (3 != 4.0) == (4 <> 5.0) == (8 < 9)
True
>>> True and False
False
>>> not (False or True or False)
False
>>> 9 < 8 and ((9/0) == 0)
False
```

Notice in the last example that the and operator does not get evaluated in its entirety. If the first argument to this operator is false, the second argument is never evaluated because the overall and statement can never be true. Because of this, the invalid operation 9/0 is never evaluated. This same short-circuiting principle holds for or as well, in the case that the first operand is true.

### 3.3 Conditional Statements

Conditional statements are based on if, elif, and else. For some Boolean statements condition1 and condition2 that have some value of True or False, the basic syntax is:
if condition1:
    #do some action
elif condition2:
    #do some other action
else:
    #do yet another action

Thinking about the operation of such conditional code is relatively straightforward. If condition1 is true, then do some action; otherwise if condition2 is true, do some other action instead. If neither condition1 nor condition2 is true, execute yet another action. elif statements and else statements do have the stipulation that a corresponding if statement exists. Note that an arbitrary number of elif statements can be used, as well as multiple if statements.

3.4 Using .py Files

So far, the code that has been run in the interpreter has been single lines, which are typed and then executed. With the branching into conditional statements, code now expands to cross multiple lines. While such code can be typed directly into the interpreter, doing so is difficult, slow, and tedious to edit. A far better approach is the use of a text editor to save and edit the code that is to be executed by the interpreter. To accomplish this, simply create a file named test.py or guide.py or anything_else.py, make sure the .py filetype is present, and enter code into this file. As an introductory example, consider the following code stored in a test.py file, which is itself present on the desktop.

Note that tabs in Python is important; indentations as shown must be present for correct operation.

It is also worth noting at this point that there is an established coding style for Python called pep8 detailing coding conventions for everything from indenting to naming variables. The examples in this document make an effort to comply with the pep8 standard. Further information on pep8 can be found online.

Continuing on with using .py files, note that everything nested by a tab under an if statement is the code that will get executed if the conditional for that statement is true. As an example of this write the following lines of code to a file called test.py stored on the desktop:

```python
import math as mt
condition1 = 1 > 3
condition2 = mt.sqrt(9.0) < 4
if condition1:
    print "condition1 was true."
elif condition2:
    print "condition2 was true."
else:
    print "both conditions were false."
```

To run this code simply navigate to the Desktop directory from the command line (on a Mac- or Linux-based machine, this would
be cd ~/Desktop); on a machine running Windows this, would be c:\Users\(username)\Desktop), and then run the command python test.py. The output should be condition2 was true. The code in the file is executed line by line as if it were typed into the interpreter. In this case, it is determined that condition2 is true, and the following print statement is executed.

For a Unix or Mac user familiar with the so-called “shebang” syntax (the use of #! at the start of a file) another option for running a Python file is to add a “shebang” specifying Python as the interpreter at the very top of the file. For this example, the first line then becomes #!/usr/bin/python, above the import math line. The shebang should specify the absolute path to the Python interpreter. If this is unknown, on a UNIX system this can be found by typing which python into the command line. The next step is to mark the file as executable with the command chmod +x test.py. After these steps are completed, the Python script can be executed by typing ./test.py.

Using either method of running Python code, it is relatively easy to create use cases for Python to see what works simply by typing it into a .py file and running it from the command line. For example, one can find out what happens when running a file containing:

```
x = 3.0
if None:
    x = False
elif not 3 or 4:
    x = 8
else:
    x = "hello"
print x/2
```

The output is 4. A short explanation is that the None value is relatively special; in this particular context it is not True (although interestingly it is also not False), and therefore the assignment of the value False to x never occurs, since a value of True is a prerequisite for the first conditional. As a holdout from earlier programming languages such as C, the integer value 0 is treated as the Boolean condition False, and any value other than 0 is regarded as True. The values of 3 and 4 are therefore regarded as True in the next conditional, thereby evaluating the or to True. This results in the assignment of the value 8 to x and also means that the else will never be executed since an accompanying if evaluated to True. The print statement is then evaluated, performing the division and then printing the result. Although this is an unrealistic example, it serves to show that experimentation with different use cases in Python is a quick and straightforward process.

The above example also demonstrates that the flexibility of Python is a double-edged sword. It is perfectly allowable to have a variable assigned a float value, and in the next line assign that same
variable a string. This can be very useful; it is easy to imagine a program that at times handles one output and at others may have a list of outputs, where the reuse of a variable is natural. However, there are some inherent dangers as is illustrated above. If x had been assigned the string “hello”, a runtime error would have occurred when the interpreter tried to divide the string by 2, an undefined operation.

3.5 Loops

Further constructs of Python are while and for loops. while loops consist of a single conditional statement. While this statement is true, the body of the loop will be continually executed. To see an example of this, consider the following loop:

```python
i = 0
while i < 10:
    print i
    i += 1
```

The numbers 0–9 should be printed when this file is run, stopping at 9. Once the counter is 10, i is no longer less than 10, and the loop terminates. In general, while loops are useful when the number of iterations of a loop is not known. For example, consider finding the first 10 odd numbers in a randomly generated sequence of numbers. It is not known at runtime how many numbers must be iterated over before 10 odd numbers are found. In a case like this, a simple counter for the number of odd numbers found and a while loop with a conditional on the counter that terminates when 10 have been found is an appropriate and natural solution.

In contrast to a while loop, a for loop is used in cases when the number of times the loop will run is known. As a basic example, the counterpart to the while loop above is:

```python
for i in range(10):
    print i
```

`range(x)` simply generates all numbers from 0 – \(x - 1\). For each of these numbers, the body of the loop is executed once. Although this example is basic, for loops in Python are actually quite powerful, with a syntax demonstrated later that lends a high degree of readability to the code.

3.6 Lists

A large part of the power of looping in Python is found in the ability to iterate over lists. Lists in Python are simply collections of integers, floats, strings, or any more complex object. Lists are created with the \([]\) syntax. Some examples include:

```python
[1, 2, 3]
[“one”, “two”, “three”]
[[1, 2, 3], [4, 5, 6]]
[“one”, 2, “three”, 4.0]
```

The second to last example above is a list of lists, a list where each object is itself a list, which can be thought of as a matrix when
the entries are numeric as above. The last example demonstrates the fact that a list can contain any number and type of object, in this case mixing strings, integers, and floats.

Accessing the elements of a list is straightforward. Singular elements of a list can be accessed and modified, as well as sublists of a larger list. It is important to remember that in Python and many other languages, the first element of a list is indexed by 0, the second by 1, and so on. Individual list elements are accessed with the list[index] syntax, whereas slices of lists, or sublists, are called with the list[lower_bound:upper_bound] syntax. Every element of the list from the lower_bound up to one less than the upper_bound will be obtained. Some examples of accessing, slicing, and modifying lists in the interpreter include:

```python
>>> L = ["one", 2, "three", 4.0]
>>> L[0]
"one"
>>> L[3]
4.0
>>> L[0:2]
["one", 2]
>>> L[1:2]
["one", 2]
>>> L[1:-1]
[2, "three"]
>>> L[:1] = [4, 5, 6]
>>> L
[4, 5, 6]
>>> L[2] = 7
>>> L
[4, 5, 7]
>>> L.append(21)
>>> L
[4, 5, 7, 21]
>>> len(L)
4
```

It is important to remember that slicing is up to one less than the upper bound, just as range(x) generates each number from 0 up to (x-1). The negative index specifies an index from the end of the list, where -1 itself specifies the last element. The append function on a list adds an element to the end of a list. The len() function is a built-in function in Python that returns the number of items in an object. The len() function can also be used for many other objects, including tuples and sets.

### 3.7 Tuples

Tuples in Python are very similar to lists, except in the fact that elements of a tuple cannot be modified once the tuple has been created, meaning tuples are immutable. This is useful to prevent errant modification in a program, which is an issue that can arise
with the usage of lists. Tuples are created with the () syntax, similar to the [] for a list:

```python
>>> t = ("one", 2, "three", 4.0)
>>> t[0]
"one"
>>> t[3]
4.0
>>> t[0:2]
["one", 2]
>>> t[:2]
["one", 2]
>>> t[1:-1]
[2, "three"]
>>> t[0] = 6
TypeError: 'tuple' object does not support item assignment
```

### 3.8 Sets

Sets in Python are another object that share many similarities with lists. The principle differences between a set and a list are that a set is unordered, it can only contain one instance of any object contained in the set, and access to an element is comparatively fast. Elements of a set must be added individually or converted from a list or a tuple. Sets are specified with the set() syntax, so, for example:

```python
>>> s1 = set()
>>> s1.add(3)
>>> s1.add(2)
>>> s1.add(1)
>>> s1
set([1, 2, 3])
>>> s1[0]
TypeError: 'set' object does not support indexing
>>> s2 = set([3, 4, 5])
>>> s2.union(s1)
set([1, 2, 3, 4, 5])
>>> s2.intersection(s1)
set([3])
>>> help(set)
```

As in the code above, sets can be thought of in much the same way as the mathematical notion of a set, and many operations exist that reinforce this treatment, as is shown by the help(set) output.

### 3.9 Loops, Continued

With the above-mentioned data structures, the power of the for loop becomes apparent. Consider the following code:

```python
numbers = set((1, 2, 3))
total = 0
for number in numbers:
    total += number
print total
```
This code is readable and structured naturally through white-space, and its purpose is apparent. When run, the output is the expected value 6. For any iterable object, be it a list, tuple, set, or some other object, the for x in y syntax provides a natural way to iterate over every item in the object.

3.10 List Comprehensions

Now that loops have been properly introduced, a more advanced, succinct, and readable way to create lists known as list comprehensions has become accessible. As a basic example, consider the problem of putting the numbers 0...9 into a list. An approach may be something like:

```python
L = []
for i in range(10):
    L.append(i)
```

The alternative offered by list comprehensions is the notation:

```python
L = [i for i in range(10)]
```

Conditionals are also usable in list comprehensions. Suppose there is a set of numeric tags stored as strings, and the goal is to obtain a list containing only the tags that start with some identifier 134. Using a built-in function of strings, a list comprehension could be used as follows for some tag_set:

```python
iden_134 = [tag for tag in tag_set if tag.startswith('134')]
```

List comprehensions can be arbitrarily nested as is needed. This can become a complicated topic to reason about, but as a basic example, consider creating simple array of numbers, a list of lists. For a 110x100 array, this can be accomplished like:

```python
array = [[num for num in range(100)] for num in range(110)]
```

Naturally, each list comprehension can become more complicated with conditional statements and different objects. In practice, although single list comprehensions typically make for more readable and understandable code, nesting list comprehensions tends to accomplish the opposite.

3.11 Dictionaries

At a high level, dictionaries store key-value pairs, where each key and value is an object. The value can be obtained by accessing the key in the dictionary, much like looking up a word in a real dictionary yields the word’s definition. Although they may not look it initially, dictionaries in Python are one of the most powerful and natural tools for solving problems.

Creating dictionaries in Python can be done either through direct creation, or through the use of built-in functions, and can be accessed through the [] syntax. For example:
As is shown, lists of both the keys and values can be accessed from the dictionary, as well as key-value pairs in the form of the dictionary.iteritems(). The dictionary.get(value, default) is a means of accessing the value associated with a key, except that instead of an error if no value exists, a predefined default value can be supplied.

3.12 Functions

Now that the basic types have been explored, it is time to define functions, which have been in use throughout this tutorial. Functions are simply snippets of code to perform some task on a variety of inputs, made to be reusable to avoid code duplication. For example, the previous len() function takes as input some object and returns the number of items in that object. The print function simply displays a string on the console. Note that print is unique for the lack of parenthesis required. Creation of functions is prompted through the def keyword, as demonstrated below in a basic implementation of the built-in len(), and values are returned from functions with the return keyword:

```python
def length(some_object):
    length = 0
    for item in some_object:
        length += 1
    return length
```

One may try this using the command-line. After importing the above code in the Python interpreter using the from filename import length syntax, the function length() should work for objects such as lists and sets, although it certainly isn’t as robust as the built-in len function. For a more interesting example, consider the task of counting the number of occurrences of each individual nucleotide in a DNA string and then printing the results. One way to accomplish this task would be:
def count_nucleotides(DNA):
    counts = {}
    for letter in DNA:
        if letter == 'A':
            counts['A'] = counts.get('A', 0) + 1
        elif letter == 'C':
            counts['C'] = counts.get('C', 0) + 1
        elif letter == 'T':
            counts['T'] = counts.get('T', 0) + 1
        elif letter == 'G':
            counts['G'] = counts.get('G', 0) + 1
    for nucleotide, count in counts.iteritems():
        print (nucleotide, count)

After adding this to a file, import it into a Python session. Once count_nucleotides has been called on some DNA string, it should successfully print the number of occurrences of each individual nucleotide. This function does not have a return value, which is allowable in Python. Functions can perform tasks without returning anything, such as printing items to the screen.

There is also a shorter way to accomplish the above task. In this particular case, a built-in function conveniently called count for strings (and lists) will count the number of occurrences of an input item in the string or list. Using this, and adding support for RNA, the following would be a more reusable function:

def count_nucleotides(string, nucleotides):
    for nucleotide in nucleotides:
        count = string.count(nucleotide)
        print ('{}{}'.format(nucleotide, count))

This function now requires two inputs, the first a string with the nucleotides to be counted and the second an iterable object containing the nucleotides of interest, such as a list, tuple, or set. Now, to call, simply import into a Python session, and then call something like:

count_nucleotides('AACCTTGG', ['A', 'C', 'T', 'G'])

Now, suppose that in 90% of use cases, the string of interest is going to be DNA, and only occasionally will RNA or some other string be input. This is an ideal case for the usage of default arguments, where an input to a function has some default value unless otherwise specified. The nucleotide counting function would now be:

def count_nucleotides(string, nucleotides=['A', 'C', 'T', 'G']):
    for nucleotide in nucleotides:
        count = string.count(nucleotide)
        print ('{}{}'.format(nucleotide, count))

    for nucleotide in nucleotides:
        count = string.count(nucleotide)
        print ('{}{}'.format(nucleotide, count))
The usage would now be:

```python
>>> count_nucleotides('AACCUUGG', ['A', 'C', 'U', 'G'])
(A, 2)
(C, 2)
(U, 2)
(G, 2)
```  

```python
>>> count_nucleotides('ACTGGCAT')
(A, 2)
(C, 2)
(T, 2)
(G, 2)
```  

### 3.13 Sorting

Functions and list comprehensions from the previous section become particularly useful in practice in the context of sorting. Basic sorting can be accomplished through the built-in sorted function. For example, after importing the random module which among other things includes a function to shuffle a list in place:

```python
>>> import random as rand
>>> l1 = [i for i in range(10)]
>>> l1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> rand.shuffle(l1)
>>> l1
[0, 6, 1, 2, 9, 3, 7, 5, 8, 4]
>>> l1 = sorted(l1)
>>> l1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l1 = sorted(l1, reverse=True)
>>> l1
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

In the last example, the default argument to the reverse argument of sorted is False; to reverse the list this argument must be explicitly defined to be True.

For a more complicated example of the usage of sorted, suppose there is an array of numbers sizes corresponding to sample sizes for some procedure, in which the entry in position 0 of the array is the size of the sample with id 0, the entry in position 1 is that of the sample with id 1, and so on. The goal is to create a function that extracts a list containing the sample ids of the largest samples by sample size. The position in the array is meaningful, so sorting the array will rearrange which sample size is contained in a particular index of the array. To preserve the association of the original array index (the sample id) with the sample size during the rearrangement of sorting, the enumerate function can be used. This function simply provides an association between the index of an array and the array element as a tuple, for example:
l = ['hydrocarbon', 'lipid', 'microbiology']
for index, word in enumerate(l):
    print (index, word)

Using this tool, to extract the n largest samples, the following function can be used:

def top_n(sizes, n):
    return [id for id, size in sorted(enumerate(sizes),
        key = lambda item: item[1], reverse=True)][:n]

This is a complicated one-line function, but it can be broken down into elements from above, and it is worth examining closely as a check for understanding. Within the sorted function, the key argument is a function that is applied to each element of the list before it is sorted. In this case, it is used to designate that it is the second part of the (id, size) tuple that is to be used for comparison. Without this argument, sorting would have been performed based on the ids rather than the sizes of the samples.

The lambda expression is simply an inline function. From the above sorted example, the equivalent function to the given lambda expression would be:

def grab_second(some_tuple):
    return some_tuple[1]

def top_n(sizes, n):
    return [id for id, size in sorted(enumerate(sizes),
        key = grab_second, reverse=True)][:n]

For extremely simple functions that do not need to be reused, a lambda expression provides a means to quickly implement the desired functionality. The second example is useful to note that functions can be passed as parameters to other functions too, which can allow for greater abstraction in code.

3.14 File I/O

File input and output in Python is also closely related to lists. Files in Python are very easy to work with. For a text file file.txt that is stored in the same directory as the Python code, the following will open the file and print line by line:

with open('file.txt') as text_file:
    for line in text_file:
        print line

As can be seen, an open file is manipulable in Python essentially as a list of strings, where in each line new line character designates the separation between entries in this list. The keyword with prompts the interpreter to properly handle the opening and closing of the file behind the scenes.

Files can be written equally simply, simply by opening them in write ('w') or append ('a') mode. If there is a second file file2.txt that needs to be written, the following will write every other line of file.txt to file2.txt:
with open('file.txt') as read_file, open('file2.txt', "w") as write_file:
    to_write = True
    for line in read_file:
        if to_write:
            write_file.write(line)
            to_write = False
        else:
            to_write = True

3.15 Testing

Testing is a critical component of the process of writing code. The above examples were relatively simple and straightforward, and so their correctness is readily verifiable through informal testing, such as printing intermediate results and checking output. As a project grows though, it is crucial that tests exist to insure the correctness of code, especially through revisions. Further, as the example in Sect. 3.4 illustrated, Python’s flexibility can be as much an asset as a danger, further prompting the need for tests. Basic testing can be accomplished quite simply; consider the grab_second helper function from Sect. 3.13, modified to be a little more robust:

```python
import sys
def grab_second(some_tuple):
    if len(some_tuple) >= 2:
        return some_tuple[1]
    else:
        return None
def test_two_tuple():
    two_tuple = ('one', 'two')
    return grab_second(two_tuple) == 'two'
def test_one_tuple():
    one_tuple = ('one',)
    return grab_second(one_tuple) == None

if 'test' in sys.argv:
    if not test_two_tuple():
        print "Error with grab_second on tuple of length 2"
    if not test_one_tuple():
        print "Error with grab_second on tuple of length 1"
    print "Tests completed."
```

sys was imported so that the argv list can be accessed. Argv is simply a list of strings of arguments passed in on the command line. So if this file is test.py, and the command to run it was python test.py, then argv is ['python', 'test.py']. In this way, tests can be run conditionally in the case that one of the arguments passed in on the command line is 'test'. To run the tests with this example then, the command becomes python test.py test. Argv then contains the string 'test', the tests are executed, and any corresponding outputs are printed.
Python also has by default a testing framework called unittest, which can be imported to automate and expand upon the above process. This can be found in the online Python documentation [5].

4 Practical Usage

4.1 File Manipulation

The basic elements of Python have been explained and demonstrated. In some cases the interweaving between pieces is shown; however, it is often best to see the language in use as a whole to understand the implementation of the various components. To demonstrate the functionality of the language in actual usage, we will consider the common task of manipulating a text file to extract and reorder information. In this particular example, there are two input text files. The first contains output presenting a simplified set of sequencing data inspired by the output of the sequencing tool USEARCH [3]. This tool groups gene sequences into the bacterial genome from which they originate. The first column of the data is the gene sequence ID, for say gene1, gene2, gene3, and so on. The second column is the bacterial genome to which that gene was assigned by the sequencer. This file appears something like:

<table>
<thead>
<tr>
<th>gene Sequence ID</th>
<th>Bacteria Genomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>gene1</td>
<td>bacterium1</td>
</tr>
<tr>
<td>gene2</td>
<td>bacterium2</td>
</tr>
<tr>
<td>gene3</td>
<td>bacterium2</td>
</tr>
<tr>
<td>gene4</td>
<td>bacterium1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The second file is simply a file containing all possible gene sequence IDs, since some gene sequence IDs may not have been assigned to a bacterium by the sequencer in the first file. This second file appears like this:

<table>
<thead>
<tr>
<th>gene1</th>
</tr>
</thead>
<tbody>
<tr>
<td>gene2</td>
</tr>
<tr>
<td>gene3</td>
</tr>
<tr>
<td>gene4</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

In the format in which these two files exist, it is unclear how many times each gene appears for each bacteria. It would be useful to use the above data to create a third file that has the data reorganized to display an overview of individual gene counts for each bacterium, such as:

| Bacteria Genome | gene1 | gene2 | gene3 | gene4 | ...
|-----------------|-------|-------|-------|-------|-------|
| bacterium1      | 1     | 1     | 3     | 0     | ...
| bacterium2      | 5     | 0     | 2     | 5     | ...
|                  | ...   | ...   | ...   | ...   | ...   |
To start, it would be useful to have a list in Python of each possible gene. This information is available from the second file, which will be called gene_file. So, a function that reads in a file of this format and outputs a list of the genes is a good starting point. Assuming the input to the function is an already opened gene_file, such a function could be:

```python
def get_genes(gene_file):
    return [line.strip() for line in gene_file]
```

All the above function does is, via a list comprehension, go through each line in the now open file, remove any trailing whitespace (in this case new line characters), and then save the resulting gene as a string in a list.

Now that the entire listing of genes is obtained, the counts of each gene for each bacterium need to be found and stored in an appropriate data structure. Ideally, this would have each bacterium associated with the counts of genes for that particular bacterium. A dictionary is the perfect data structure for this purpose. The names of the bacteria from the second column of the first file, called bacteria_file, will serve as the key, and the counts of each gene appearing with these bacteria will be the values. In this way, each line of the bacteria_file can be read in, and then the gene count of the appropriate bacterium can be updated, line by line.

As for storing the actual counts, Python once again provides a built-in means of handling this common task, in the form of a counter object. This object must be imported, but once that is accomplished, it provides an easy way to track counts. Queries to a counter object are performed with the [ ] syntax, much like a key lookup with a dictionary. If the query item exists, the count is returned. If the query item does not exist, the count defaults to return 0. Counts can be added and updated as is necessary in the expected way.

To summarize what now must be done, for each bacterium encountered in the bacteria_file, an entry must be made in the dictionary, where the entry is a counter object. If a counter already exists because the bacterium is in the dictionary, the preexisting counter should not be replaced by a new counter; rather the preexisting counter should be updated. Then, for the gene that is on the same line as the bacterium of interest, the bacterium’s count of that gene in its counter object should be incremented by one. At the end, all bacteria in the file will have a counter in the dictionary, and this counter will store the number of times each gene originated with each individual bacterial genome.

Code for the outline above is as follows:

```python
from collections import Counter
def get_bacteria_dict(bacteria_file):
    bacteria_dict = {} #create the overarching dictionary to store the bacteria-counter key-value pairs
```
for line in bacteria_file:
    gene, bacterium = line.strip().split(' \	')
    # remove the whitespace, and then split the tab delimited line into two pieces
    bacteria_dict[bacterium] = bacteria_dict.get(bacterium, Counter())  # if the bacterium already has a value in the dictionary, don't modify it. Otherwise, make the value a new counter object by specifying a default value.
    bacteria_dict[bacterium][gene] += 1  # increment the gene count in the counter object (guaranteed to exist by the previous line) by one. Remember that if the counter did not have an entry for a gene already, the gene starts at a count of 0. Therefore, after this line the count will be 1.

return bacteria_dict

Now, a function exists that takes as input the name of a gene file and outputs a list of genes, and a function exists that takes as input a bacteria file and outputs a dictionary containing all the appropriate gene counts. Using these two functions, the third file, output_file, can be written. First, the gene list can serve as the source for the header of the file. Then, for each bacterium in the bacteria dictionary output by the second function, the counter associated with the bacterium can be queried for each gene in the gene list, and this output can be written to the file, in the same order as the genes appear in the gene list. In this way, the table of appropriate counts can be created. To implement this, two functions can be written, the first to write the header and the second to write the rows:

def write_header(gene_list, output_file):
    output_file.write('	')  # provide one tab offset to align the columns of the output file
    for gene in gene_list:
        output_file.write(gene + '	')  # write each gene to the top line of the file, in a tab separated format.
        output_file.write('
')  # add a new line so that the next function will begin the rows at the proper location.

def write_rows(bacteria_dict, gene_list, output_file):
    for bacterium in bacteria_dict:
        output_file.write(bacterium + '	')  # write the bacterium name as the first element of the column, with appropriate tab placement
        for gene in gene_list:
            gene_count = bacteria_dict[bacterium][gene]
            output_file.write(str(gene_count) + '	')  # sequentially grab and write the counts of each gene for the bacterium with appropriate tab separation
            output_file.write('
')  # write a new line before the next bacterium is queried.

Finally, to tie this all together, a driver function can be used to open files and pass the appropriate parameters to functions in the correct order:
def main(gene_file_name, bacteria_file_name, output_file_name):
    with open(gene_file_name) as gene_file, open(bacteria_file_name) as bacteria_file, open(output_file_name) as output_file:
        # open all files as read only, except the output file to be written to.
        gene_list = get_genes(gene_file)
        bacteria_dict = get_bacteria_dict(bacteria_file)
        write_header(gene_list, output_file)
        write_rows(bacteria_dict, gene_list, output_file)

    To run this, one would include the above functions in one file in the same directory as the bacteria and gene files. Then, after importing the main function to an interpreter session, the code should be runnable.

    In this example, the basic types of strings and integers were combined with dictionaries, lists, and file I/O in the creation of several key loops to transform one set of data into another. These pieces ultimately form the backbone of the language. Due to space limitations the treatment of these components is necessarily incomplete. There is more functionality in lists, dictionaries, sets, and related structures that is implemented in Python. It is therefore critical for the interested reader to explore the full functionality that is offered by Python. In the development of python code, we often find that instead of first implementing our own version of a mundane task, a quick web search will reveal that this work would have been redundant. This is a common theme with Python and its rich online support ecosystem; often the applications of the language are unique, but the pieces required to implement the applications are readily available to be fit and molded to the problem at hand.

4.2 Third-Party Packages

    This notion is taken even further with third-party packages. These are collections of functionality implemented in Python that essentially expand the installed language. The handling of large arrays of numbers in Python is sometimes clunky and slow, a problem that the NumPy package [6] addresses. Critical loops in a Python file can largely determine the speed at which a file executes; in this case Cython [7] offers a means to incorporate speedy, compiled C code directly into a Python file. Biological data is best handled robustly with third-party packages as well, such as scikit-bio [8], which is itself built on top of the machine learning functionality that is implemented by scikit-learn [9]. The list of these packages goes on and on, and it is well worth the time to search and build upon these packages in one’s own work.

    These third-party packages can be installed on a machine alongside the current Python distribution, frequently using a package manager called PIP. Once installed, packages can be imported and used in much the same way as the math functionality brought in with the import math statement. If NumPy has been installed on
a machine, using the command import numpy will make the functions of the NumPy package callable in the current Python file.

4.3 Testing

Testing is a critical component of the process of writing code. The above examples were relatively simple and straightforward, and so their correctness is readily verifiable through informal testing, such as printing intermediate results and checking output. As a project grows though, it is crucial that tests exist to insure the correctness of code, especially through revisions. As the example in Sect. 3.4 illustrated, Python’s flexibility can be as much an asset as a danger, further prompting the need for tests. Formal testing exists in the form of the unittest framework.

5 Conclusion

This overview has described the basic functionality of Python, with the hope that the functionality demonstrated has illustrated the ease with which often complicated tasks can be performed using features of the language. Writing python code often involves solving problems that are very similar to or exactly the same as problems that others have solved in the past. The same is true for debugging erroneous code, because searching the Internet for examples where others have received a similar error message is often the fastest way to identify the source of the error. Therefore as developers we find that searching the internet for helpful examples and code is an integral part of the learning process in this area and can speed up the development process significantly. Time spent learning Python can be invaluable, particularly in the scientific community. It offers quick, understandable ways to solve both the common and the complex tasks that present themselves on a daily basis to a researcher in Hydrocarbon and Lipid Microbiology.

References