# Introduction to Python Programming and Project Design

Steven Lakin, DVM

2018 May 14

# Contents

## 1 The Zen of Python

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

## 2 Introduction

With the advent of big data and the need for automation of repetitive tasks, basic programming skills are becoming a necessity for working in many fields. However, much of the instructional material on programming is intended to build a very solid foundation in programming basics. For our purposes, we can skip the details of rudimentary programming and take a more hands-on approach to basic scripting, since this is much of what we as non-computer scientists will be doing. We will be using the Python language, since it is an intuitive and powerful programming language.

Python, named after the Monty Python skits, was built with the intention of being easy to use, quick to learn, and syntactically fast; this is sometimes referred to in the documentation as being "Pythonic," based on the ideals of the language. Because of this, Python is what is called a "high-level" language; much of the clunky syntax of other languages (Java, C, R) is removed in python. There are no semi-colons at ends of lines, no brackets around portions of code, or the need to pre- define variables before use. You will find that this makes Python a fast language to program in, since we don't have to worry as much about typing and checking syntax. Much of the baseline "work" of lower-level languages has been built into Python, which allows you to access Python's intuitive structures and do your work as easily and fast as possible.

In this workshop, we will be focusing on applying Python to biological problems. Much of this "patchwork" programming for solving small problems is well-suited to short segments of code called scripts. A script is simply a file of code that does something. Scripts can be combined to make programs, packages, modules, and generally "software," all of which are generally the same thing with slightly different semantics in different languages. We will mostly be scripting in this class, though we will work toward making packages and learning the basics of building more advanced code structures.

The workshop will be divided into two segments: the first will be a lecture on a programming topic, and the second will be applying those concepts to miniature problems on Rosalind, named after Rosalind Franklin, whose work in X-ray crystallography contributed significantly to the discovery of DNA structure. These problems are bioinformatics related, which is a field that combines biology, computer science, mathematics, and statistics to solve biological problems involving large data sets. You will need to make an account on Rosalind to track your progress.

## 3 Obtaining Help

Every programmer needs a reference source for learning new skills in a language or to troubleshoot problems encountered during the programming process. There are many sources of this information, including physical textbooks or manuals. However, the vast majority of programmers these days utilize online resources that have been curated and developed by the community. Probably the most popular and frequently utilized online resource is Stack Overflow, which is part of the popular Stack Exchange system of websites for community-based question and answer forums. I would recommend Googling stack exchange plus whatever keywords or error codes that you have in order to determine the best path forward in your code. Very frequently, someone has encountered the same question or problam as you, and that question will already have been answered on Stack Overflow. However, if you do ever encounter a novel problem (it has only happened to me twice in my entire programming career), then you can submit a new problem on Stack Overflow and hopefully get the problem solved. Alternatively, the online Python manual often has answers about the language itself.

# 4 Python Fundamentals

## 4.1 Installing Python

**Python Versions**
There are two main distributions of Python: Python 2 and Python 3. These days, virtually every application should be compliant with Python 3, so it's recommended that you download the most recent version of Python 3. The difference between the two is minimial from a basic user perspective, however some of the "grammar" or syntax differs between them. This workshop will be conducted as if everyone has Python 3.

**Python Download**
You can download Python for Windows and MacOS by visiting the Python download page. Linux users can download Python through their appropriate package manager. I will assume if you're using Linux that you know the basics of the operating system.

**Integrated Development Environments (IDEs)**
When we get to the point where we are writing scripts, you may find it helpful to download specialized software called an IDE, which is essentially a text editor that knows the Python syntax, can highlight regions of code, helps with code completion, and can help debug programs. There are a wide array of IDEs for Python, some more feature-complete than others. I prefer to use a more complete IDE called PyCharm (Jetbrains), however some people prefer lighter IDEs, and others prefer to use a standard text editor like NotePad that has no coding-specific features. You can research a few of these IDEs and decide what is the right fit for your coding experience.

**Python Packages**
Python has both core functionality and the ability to be extended for other purposes through the development of packages. Packages are modules that perform a specific function, such as helping to scrape websites, download files, multiply matrices, and visualize data through graphing/plotting. You can obtain these packages through Python's package manager called Pip. A basic tutorial on how to use Pip to obtain packages can be found in the Python user manual.

## 4.2 Variables and Basic Operations

As a Python programmer, you'll encounter two situations commonly while coding: testing small bits of code in real time, and assembling larger pieces of code into a program or script. For testing code in real time, you'll use the **Python Interpreter**. To get to the Python Interpreter, double click on the Python icon or open a command-line terminal and type python in lower-case. The interpreter will have three greater-than signs as its prompt, like so:

```
>>>
```

The Python Interpreter has all of the functionality that we will be using later, however it is difficult to program larger pieces of code in the Interpreter, so we will be switching to scripting later on. The Interpreter can perform basic arithmetic such as a calculator would do; simply type the equation and press enter:

```
>>> 3 - 5
-2
>>> 3 + 5
8
>>> 3 * 5
15
>>> 3 / 2
1.5
>>> 3e5
300000.0
```

Notice that all input either starts with the three greater-than signs or three periods, while all output simply is on a blank newline. While performing basic arithmetic operations is a nice feature of Python, we could simply use a calculator for this. Python's functionality and power comes from being able to store values in **variables**:

```
>>> a = 3
>>> b = 5
>>> a + b
8
>>> a * b
15
>>> a * 5
15
```

For those interested in proper programming terminology, we refer to variables as being a **left-hand values** or l-values, since it is on the left hand side of the equation, while **right-hand values** or r-values represent the values that the variable will take. Remember that right-hand values **get assigned to** left-hand values; you cannot interchange the two.

In Python, we are not limited to numbers. Python, being a simple and elegant language, has only a few **types** of values. We have already encountered the **Integer** type and the **Float** type: integers referring to the mathematical definition of whole real numbers, and floats referring to decimal-point real numbers. A commonly-used third type is the **String** type, which holds words. Less commonly used but important is the **Boolean** type, which indicates one of two values: true or false (1 and 0, respectively in binary). Python often will automatically convert between types depending on what you need to do:

```
>>> a = 3
>>> b = "1"
>>> c = "String Example"
>>> print(a, b)
```

```
3 1
>>> c + b
'String Example1'
```

However, notice that Python will not automatically convert strings into integers or floats; to do that you'll have to **coerce** the value that you know complies with the new type into that type, like so:

```
>>> a = "1"
>>> b = 3
>>> a + str(b)
'13'
>>> int(a) + b
4
>>> float(a) + b
4.0
>>> bool(a)
True
```

Notice that the addition sign has different functionality depending on the type for which it is used: numbers are added together, while strings are **concatenated** in the order of which they were added. Other functionality that are built into Python include the native **functions** that you can use for common operations like printing text to the terminal:

```
>>> myvariable = "Hello World"
>>> print(myvariable)
'Hello World'
```

Functions take variables or values as input and are used with parentheses as above. Next, we will learn how to create our own functions as well, which is the foundation of programming.

## 4.3 Functions

Functions are the building blocks of programming; each function should have a single purpose, be specific, and have an easily-interpretable and appropriate name. The parts of a function include the **name**, the **arguments**, the **function body** where the code resides, and an optional **return value**. Functions in Python are **defined** like so:

```
>>> def function_name(argument1, argument2):
...     print(argument1)
...     print(argument2)
...     return argument1 + argument2
...
```

The above code defines the function called $function\_name$, which takes in two arguments, "argument1" and "argument2." It first prints each argument on a separate line, then returns their sum. By default, return values get printed to the Interpreter if they are not used. Alternatively, we can treat return values as right-hand values and assign them to another variable:

```
>>> function_name(1, 2)
1
2
3
>>> newvalue = function_name(1, 2)
1
2
>>> print(newvalue)
3
```

Notice that in our function definition, the function body (including the return statement) are indented; in Python, all code that falls into a **code block** such as a function needs to be indented the same number of spaces or tabs. The debate between the use of spaces or tabs is semantic but hotly debated amongst Python programmers. I personally prefer tabs, as they take fewer keystrokes, however you can use what you wish as long as you are consistent. Each level of indentation indicates another **nested** code block. We will clarify this in the following sections once we have the ability to nest code blocks.

Functions can take two kinds of arguments: **positional** and **optional** arguments. Positional arguments are mandatory when using the function and are always placed before optional arguments. Optional arguments can be placed in any order and have a default value explicitly specified in the function definition. There are various reasons to use positional versus optional arguments; perhaps your function will not work without a certain number of arguments, like the function we made above. Alternatively, if you want to specify a default value but still allow other values, you would use an optional argument. Here is a function definition that includes both positional and optional arguments:

```
>>> def add_and_multiply(summand1, summand2, coefficient=1):
...     return coefficient * (summand1 + summand2)
...

>>> add_and_multiply(2, 3)
5
>>> add_and_multiply(2, 3, 3)
15
>>> add_and_multiply(2, 3, coefficient=3)
15
>>> add_and_multiply(summand1=2, summand2=3, coefficient=3)
15
```

Arguments can be called by name or by position, however if you are striving to have legible code, it usually pays off to explicitly spell out the arguments so others (or you several years later) can read the code and understand your intentions.

With functions and variables, you have the most basic use cases of Python covered and can begin performing operations on input values. However, often times we want to store larger amounts of data than single values, and often those values need to be stored in a way that relates them to one another or orders them in a certain way. To do this, we will need to learn how to use the **data structures** that are inherently available in Python. However, we will first practice what we have learned so far on one of Rosalind's introductory bioinformatics problems: counting nucleotides.

**Rosalind Problem 1: Counting Nucleotides**
Click here to visit this problem on Rosalind

Often times we want to obtain information from string-based data, such as words, paragraphs, or in this case nucleotides from a DNA sequence. In this problem, you will use functions inherent to the **String type** to count the number of adenine, cytosine, guanine, and thymine nucleotides in a DNA string of variable length. To do this, make use of the **count** function that is inherent to the String type, and define a function that prints out the number of ACGT nucleotides in that order:

```
>>> dna = "ACGTGTGTGCCCGTGA"
>>> dna.count("A")
2
>>> dna.count("C")
4
```

## 4.4 Data Structures

Data structures are used to store information in a group or organized context; each kind of data structure was designed in Python with a certain use in mind, so each has its advantages and disadvantages. One of the most common uses of data structures in coding is to store multiple values in a specific order. The ordered data structures in Python include the **list** and **array**. The primary differences between lists and arrays is that values within a list can be modified after creation while arrays cannot; this translates into a speed advantage computationally when using arrays, since the computer knows they won't be modified after the fact. However, for the majority of applications, lists will be what you choose to use, as often we want to retain the ability to modify the values of the list.

Lists in Python are defined using square braces [ ], and each element is associated with a position in the list. These positions **start at 0** for the first element; in computer science, this is referred to as a **zero-indexed** language. Here is an example of a list and how we can access each element of the list using zero-indexed integers:

```
>>> mylist = ["one", "two", "three"]
>>> mylist[0]
'one'
>>> mylist[1]
'two'
>>> mylist[2]
'three'
```

Here, the variable $mylist$ stores the list, and we access each element of the list using square braces immediately following the variable name. In Python, this is called **slicing** the list, since the slicing notation can do more than just access a single value at a time; we can obtain multiple values in a chunk by using the inherent slicing notation:

mylist[start:stop:by]

```
>>> mylist = [1, 2, 3, 4, 5]
>>> mylist[0:3]
[1, 2, 3]
>>> mylist[0:5]
[1, 2, 3, 4, 5]
>>> mylist[3:5]
[4, 5]
>>> mylist[::-1]
[5, 4, 3, 2, 1]
>>> mylist[::2]
[1, 3, 5]
>>> mylist[::3]
[1, 4]
```

Here, we access chunks of the list by using the start:stop notation, where the start is the index of the value we want to start at, and the stop is the index **of the element to the right** of where we want to stop. It may be easier to visualize it like so:

$$[{}_0 1,_1 2,_2 3,_3 4,_4 5_5]$$

In the example above, the subscripts represent the slicing index positions, so to obtain the numbers 1, 2, 3, we would slice from 0 to 3. The optional third command in the slicing notation is the **by** notation. This returns elements **by every X index**, so for instance if we wanted every other number in the list, we would slice the whole list by 2, as in the code block above: mylist[::2]. Leaving a start or stop field empty is shorthand notation for "the beginning of the list" and "the end of the list," respectively. If you provide a negative number in the by position, this reverses the list and otherwise behaves the same as the normal by notation. Finally, we can add elements to the list using two different methods: the **addition operator** and the **append function**:

```
>>> mylist = []
>>> mylist + [0, 1]
[0, 1]
>>> mylist += [0, 1]
>>> mylist
[0, 1]
>>> mylist.append(2)
>>> mylist
[0, 1, 2]
```

Pay close attention to the second and third line of code above: in the first with the addition operator alone, we simply printed out the 0 and 1 added to the list, but we didn't permanently modify the list. To permanently add the elements to the end of the list, we had to use the addition operator in combination with the equals sign operator, a construction that is commonly referred to as an **increment** operation. For completeness, we will digress momentarily to show that this increment operation can be performed also on other data types and structures, such as integers and arrays:

```
>>> a = 1
>>> a += 1
>>> print(a)
2
>>> myarray = ("one", "two")
>>> myarray += ("three", )
>>> myarray
('one', 'two', 'three')
```

The **array** data structure is defined by parentheses as seen above and operates similarly to the list, however we cannot modify the values of an array; the property of lists that allows them to be modified is called **mutability**, and we call lists **mutable** while arrays are **immutable**:

```
>>> mylist = [0, 1, 2]
>>> myarray = (0, 1, 2)
>>> mylist[1] = 3
>>> mylist
[0, 3, 2]
>>> myarray[1] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

In Python, arrays are technically called **tuples**, however the vast majority of other languages call their similar data structure arrays, so we will use that terminology here for convenience. There are physical differences in how lists and arrays interface with your computer that give them the properties that define them; we will cover these semantics later in a section that discusses what a computer actually is and how basic operations are performed. For now, we still have two important data structures to cover: the **dictionary** and the **set**, which are the two native **unordered** data structures.

When order of the values in a group does not matter, we can use the unordered data structures. The first, called a **set**, can be thought of the same way as the mathematical definition of a set: a group of things, whether that be strings, integers, floats, or even other data structures like arrays or lists. A set is useful because **checking for membership is very fast**. We will digress into some basics of computer science algorithms later in the course, however for now know that the use case of a set is **to collect items** and **to check if an item is in the set**. Sets are defined explicitly by name in Python, and we add elements to them with the **add** function. We can then check for membership by asking if a value is in the set:

```
>>> myset = set()
>>> myset
set()
>>> myset.add(1)
>>> myset.add(2)
>>> myset
{1, 2}
>>> myset.add(1)
>>> myset
{1, 2}
>>> 1 in myset
True
>>> 3 in myset
False
```

Notice that sets don't store duplicate values; when we try to add a duplicate value to the set, the set automatically checks for membership first and only adds the value if the value is not yet present in the set. This is useful for collecting unique values. While sets are quite useful, a more common data structure that we will need to use is called the **dictionary**.

Dictionaries **relate values together** and are often called **associative arrays** in other programming languages, since they associate a **key** with a **value**. These associations are often called **key-value pairs** and are extraordinarily common in basic programming tasks. Let's say we have names and phone numbers and we would like to relate them together; we can do this by creating a dictionary. Note that dictionary keys must be unique, while values can be duplicated. We commonly add to dictionaries using two methods: the **index operator**, and the **update** function; however, there are more ways to add to dictionaries if you're interested in reading the Python documentation.

```
>>> mydict = {"John Doe": "970-555-0001", "Jane Doe": "970-555-0002"}
>>> mydict
{'John Doe': '970-555-0001', 'Jane Doe': '970-555-0002'}
>>> mydict["James Smith"] = "970-555-1234"
>>> mydict
{'John Doe': '970-555-0001',
'Jane Doe': '970-555-0002',
'James Smith': '970-555-1234'}
>>> mydict.update({"Jane Smith": "970-555-1235"})
>>> mydict
{'John Doe': '970-555-0001',
'Jane Doe': '970-555-0002',
'James Smith': '970-555-1234',
'Jane Smith': '970-555-1235'}
```

Notice that the keys are displayed left of the colon and the values with which they are associated are displayed on the right side of the colon. We can also access lists of the keys and values separately with the **keys** and **values** functions, respectively. Checking for membership is also useful, and by default we check for membership in the keys, since they are unique, however we also can force Python to check for membership in the values by explicitly stating that:

```
>>> mydict = {1: "red", 2: "blue", 3:"green"}
>>> mydict.keys()
dict_keys([1, 2, 3])
```

```
>>> mydict.values()
dict_values(['red', 'blue', 'green'])
>>> 1 in mydict
True
>>> 4 in mydict
False
>>> "red" in mydict
False
>>> "red" in mydict.values()
True
```

There are many more aspects to dictionaries that we will not cover here for brevity; I suggest if you're interested in the versatility of dictionaries to read the Python documentation on them in detail. For now, simply know that dictionaries store key-value pairs, and that such a data structure is highly useful in many programming contexts. We will see concrete examples of why this is the case in the Rosalind problems.

As an aside, there are many other data structures available in other Python packages, but with these four core data structures, you're ready to start learning the truly useful part of programming: **logic** and **flow**. There will be no Rosalind problem for this section, since we have to learn the next topics to do anything intermediate in programming.

# 5 Logic and Control of Flow

## 5.1 Conditional Operators

Logic is a foundational concept in computer science and programming; when we encounter certain values, we often want to do something or store those values differently depending on some condition. Perhaps we see a value that is interesting, so we want to keep it, however we don't want to keep uninteresting values. Perhaps we only want to add an element to a list if it isn't already in the list (though if you remember last section, you could use a set for this instead!). Logic in Python is very simple compared to other language, and the developers of Python have tried to make conditional logic statements match what you are thinking in your head when you're working on a problem. They also support the standard language conditional statements and operators as well, if you're not comfortable using the Pythonic approach.

Perhaps the most simple conditional statement is the **in** statement; this checks for membership in a data structure for a value or even another data structure. The **in** statement has some computational cost depending on the data structure, but it can be used with almost anything:

```
>>> mystring = "ACCGTG"
>>> "A" in mystring
True
>>> mylist = [1, 2, 3]
>>> 2 in mylist
True
>>> 2 not in mylist
False
```

Notice two aspects about the above code: to check for membership, we use **in**, and to check for the opposite condition, we simply add a **not** in front of the **in** statement. Secondly, **conditionals return boolean values**, either True or False. These values can then be used to create logic that performs certain code on some values and not others. If anyone is nerdy like me about computer science topics and wants to know where this idea originated, you can follow this link to learn more about logic gates.

The following are the **Python conditional operators** that we can use to **compare** values to one another: **is, ==, is not, !=,** $>$, $>=$, $<$, $<=$.

```
>>> 2 > 3
False
>>> 3 >= 3
True
>>> 2 is 3
False
>>> 2 is not 3
True
>>> 2 == 2
True
>>> 2 != 2
False
```

Again, each of these operators returns a boolean value; these boolean values will often be used by the statements in the next section, called **if statements** that will allow you to conditionally perform certain blocks of code.

## 5.2 If Statements

If statements begin with an **if**, include a conditional statement and operator, have some code body similar to a function, and will optionally include additional statements and conditionals with one or more **elif** (else if) statements and optionally ending with an **else** statement. Here is an example:

```
>>> a = 4
>>> b = 3
>>> if a > 4:
...     print("a greater than 4")
... elif a > b:
...     print("a greater than b")
... elif b > a:
...     print("b greater than a")
... else:
...     print("Don't know")
...
a greater than b
```

Notice how we maintain the use of indentation to describe blocks of code for the body of statements, exactly the same as for function definitions. There can be as many lines of code as are needed between these statements, as long as they all have the same indentation. You can also nest if statements with an additional level of indentation:

```
>>> a = 4
>>> b = 3
>>> if a is 4:
...     if b is 3:
...             print("a = 4 and b = 3")
...
a = 4 and b = 3
>>> if a is 4 and b is 3:
...     print("a = 4 and b = 3")
...
a = 4 and b = 3
```

The **and** conjunction can be used to chain together conditionals, or you can simply nest them; either way is equivalent, however usually programmers consider the nesting of compound logicals to be redundant and unnecessary. To reiterate, if-elif-else statements must begin with an if statement, elif and else are optional, and there can be multiple elifs but only one else. We will be using logicals quite frequently in our future Rosalind challenges. Now, onto the workhorse of programming: loops.

## 5.3 For Loops and Comprehension

The reason we store values in data structures is typically to use them later, and when we're dealing with thousands or millions of values, we can't reasonably access them one at a time. Loops allow us to **iterate** over a large number of values *very quickly* and perform some manipulation on them. Perhaps it will be to generate and fill new lists or to read in a large file and iterate over its lines looking for or storing information as we go, but loops will always be the solution to repetitive tasks in programming. The flow statement for loops is **for**, used as so:

```
>>> for i in range(1, 10):
...     print(i)
...
1
2
3
4
5
6
7
8
9
```

The overall construction for a **for loop** is "for variable in object/iterator:", where we are either looping over some data, perhaps stored in a list or dictionary, or we are generating new data in the loop, such as in the example above. The **range** function creates a set of numbers between the start and stop arguments, in this case 1 and 10; we can then use this object as the basis for our for loop. We technically don't have to use the variable (in this case i); if we simply wanted to print "Hello world" 9 times, we could instead do that:

```
>>> for i in range(1, 10):
...     print("Hello world")
...
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
```

Of course, the most commonly used construction of loops is to iterate over some data structure, such as a list or dictionary. Here, we will create a list with several elements, iterate over them, and find a specific value and print it out to the Interpreter.

```
>>> mylist = ["Arthur", "Lancelot", "Gawain", "Galahad"]
>>> for knight in mylist:
...     if knight is "Arthur" or knight is "Galahad":
...             print(knight)
...
Arthur
Galahad
```

You can start to see the general pattern of programming at this point: we have some data, store that data in the appropriate data structure, manipulate that data depending on our task, and output something that we need. Generally, it is best practice to write **functions** that perform a specific task; we will cover coding best practices later, but this practice helps us to compartmentalize our operations and makes our code much more legible when you need to revisit it or share it later. Here is an example of a function that repeats text a certain amount of time depending on its arguments:

```
>>> def text_repeat(text, n_times):
...     for i in range(n_times):
...             print(text)
...
>>> text_repeat("Hello world", 3)
Hello world
Hello world
Hello world
```

In addition to these standard loop constructions, Python has sought to be elegant in its ability to generate data objects on the fly, so anytime we're creating a list, dictionary, or array, we also have the option of looping using **comprehension**. Comprehension is a quick, one-line way to generate data into a data structure, and it can be quite useful. Here are examples of list and dictionary comprehension:

```
>>> mylist = [x for x in range(10)]
>>> mylist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mylist = [x * 3 for x in range(10)]
>>> mylist
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]

>>> foo = [["foo", "bar"], ["bar", "foo"], ["foobar", "barfoo"]]
>>> mydict = {k:v for k, v in foo}
>>> mydict
{'foo': 'bar', 'bar': 'foo', 'foobar': 'barfoo'}
```

Now we're cooking with gas; what you have learned so far should be enough to carry you through basic scripting. There are a few more topics that may help, however, for specific situations where we need more control over our flow statements. In the next section, we will be covering a new type of loop that is used for specific cases and statements that will allow us to control flow.

## 5.4 While Loops and Control of Flow

In certain situations, we want to continue iterating indefinitely until we detect a certain value, and at that point, we want to cease iteration. This task is well-suited to either a **while** loop with a conditional or a **while** loop with a **break** statement. While loops will continue iteration until the condition is satisfied or it gets told to stop by a control of flow statement. It's easiest just to see it in action, so here it goes:

```
>>> x = 1
>>> while x != 64:
...     x *= 2
...     print(x)
...
2
4
8
16
32
64

>>> x = 1
>>> while True:
...     x *= 2
...     print(x)
...     if x == 64:
...             break
...
2
4
8
16
32
64
```

Here, we continue multiplying the value held in x by 2 and printing it until it reaches the value of 64, at which time we cease iteration. This can also be done with a conditional and break statement, as seen above. Be careful with the **while True** construction: if a break statement condition is never satisfied, you will have an infinite loop and will have to restart Python in order to stop it from looping on your computer forever. While loops can be helpful when we don't know the size or number of elements we will be handling, for instance while reading in data from a file that is too big to open all at once, so we have to iterate through it line by line until we reach the end. The other major control of flow statement is the **continue** statement, which allows us to skip an iteration if a condition is satisfied:

```
>>> for x in range(10):
...     if x > 6:
...             continue
...     else:
...             print(x)
...
0
1
2
3
4
5
6
```

All code that is after the continue statement is never reached, and we continue to the beginning of the next iteration within the loop. This can be useful if there are values we know we won't be using, so we can save time and not execute the code for those values.

And... that's really all there is to the basics of programming. Not too bad, right? Of course, we still need to transition away from using the Python Interpreter and begin to write actual scripts: the cornerstone of the Bioinformatician's toolbox. We will be covering scripting in the next section along with several common scripting tasks, such as reading and writing files from your computer. But first, we will apply the skills described in this section to a Rosalind problem.

**Rosalind Problem 2: Transcribing DNA into RNA**
Click here to visit this problem on Rosalind

Translation is an important part of bioinformatics; often times we want to translate one value into another, and we can do this in several ways. In this simple case of translating DNA nucleotides into RNA nucleotides, we can use either **string substitution** or we can use **dictionaries**, since we are essentially associating one value with another to translate it. In this problem, you will write a function to perform this translation task and apply it to the Rosalind problem. For demonstration purposes, here is a slightly related example that may help in your programming:

```
>>> example_string = "ABCDEFG"
>>> example_string.replace("C", "Z")
'ABZDEFG'
```

Using dictionaries as translation objects requires looping using either loops or using list comprehension syntax:

```
>>> example_string = "AAAACC"
>>> translator = {"A": "A", "C": "T"}
>>> "".join([translator[x] for x in list(example_string)])
'AAAATT'
```

The above code first **splits the string** by calling the **list** method, coercing it into a list where each value is a single letter. We then run each letter through the associative array (dictionary) by using **list comprehension**. The result is still a list of letters, so we have to **join** the list back into a single string by calling the **join** function with no separator. Below is the step-wise process for demonstration purposes.

```
>>> example_string = "AAAACC"
>>> translator = {"A": "A", "C": "T"}
>>> >>> list(example_string)
['A', 'A', 'A', 'A', 'C', 'C']
>>> [translator[x] for x in list(example_string)]
['A', 'A', 'A', 'A', 'T', 'T']
>>> "".join([translator[x] for x in list(example_string)])
'AAAATT'
```

# 6 Scripting

## 6.1 Elements of a Python Script

## 6.2 Coding Style and Best Practices

## 6.3 Parsing Command-line Arguments

## 6.4 Reading and Writing Files

### 6.5 Case Studies in File Parsing

# 7 Object Oriented Programming

## 7.1 Python as Objects: What is an Object?

## 7.2 Classes

### 7.3 Namespaces

## 7.4 Modules

# 8 Retrieving Data from the Internet

## 8.1 RESTful APIs and the Requests Package

## 8.2 Parsing JSON and XML data

### 8.3 Interfacing with Bioinformatics Public Resources

## 8.4 Version Control and GitHub

# 9 Computer Science Fundamentals

## 9.1 How Code Interfaces with your Machine

## 9.2 Computational Cost

## 9.3 Algorithm Fundamentals

## 10 Project Design