Programming Fundamentals for Intro to Bioinformatics

The examples in this document are written for Python 2.7.

Variables

You can store information (e.g. a number or other kind of data) in a variable. When you first use a variable, you must give it a name. You can **assign** certain values or other data to a variable by using the equal sign.

Example:

```
num_of_cats = 5
```

The variable `num_of_cats` now contains the number 5. When naming your variable, you cannot use spaces. It is also helpful to give your variables informative names, rather than just 'x' or 'y'.

Comments

You can "comment" out certain parts of your program by using the # symbol. The # symbol tells the computer running your program to ignore the rest of that line. This allows programmers to put human-readable notes in the body of their program. Writing informative and frequent comments is a good programming habit. Often times, you may find yourself returning to code that you've written too long ago to remember.

Example:

```
num_of_cats = 5
#This does nothing.
```

Print

`print` is a function built into most programming languages that will 'print' whatever you give it to terminal (the screen). Using print smartly can help you debug your program by allowing your program to give you visual feedback as it moves through its structure.

Example:

```
print 10
#This outputs '10' to your screen.
```

Expressions

An expression is a programming "phrase" that evaluates to something else. You can set variables to the result of an expression. Expressions can include functions, mathematical operators, variables, and constants.

Example:

```
a = 2+2 #This is 4
b = 12 / 4 * -1 #This is -3
c = 2 ^ 3 #This is 8
d =  a + b #This is 1
e = abs(b) #abs() is a function that returns absolute values. e is 3.
```

Functions

A function is a "mini program-in-a-program" that *takes in parameters* and *returns* some sort of output. Functions can be either built-in to a programming language or user-defined. Once we define our own functions, we can call on them whenever necessary by using their name. We can define our own function using the following format. Be sure to indent the line following `def`; this indicates that the code in the indented block is **inside** of the function.

```
def function(parameters):
    #put code here for...
    #what the function should do
    return output
```

Here is an example function that adds two numbers together and returns the result.

```
def add_two_numbers(parameter_1, parameter_2):
    my_result = parameter_1 + parameter_2
    return my_result

#Now we're going to run this function.
num1 = 7
my_sum = add_two_numbers(num1,10)
#my_sum now contains the number 17.
```

Lists

A list is exactly what it sounds like: a list of variables. You could also have lists of lists, but that gets more comlicated than necessary for this course. Creating

a list is similar to creating a variable, but you need to use commas to separate list items and square brackets to enclose your list.

Example:

```
my_fav_numbers = [1, 2, 4, 7, 88]
#my_fav_numbers is now a list containing 5 variables
```

You can access items in the list by calling them by their index (i.e. their position number) in the format `list[index]`. The position of an item in a list counts upwards from left to right, **starting at zero.** When you call an item in a list, you can treat it like any other variable.

Example:

```
my_fav_numbers = [1, 2, 4, 7, 88]
my_fav_numbers[0] #This is 1
my_fav_numbers[4] #This is 88
my_fav_numbers[4] = 9
my_fav_numbers[4] #This is now 9
```

If you want to find the length of a list, there is a built in function (see below for function explanation) that you can use called len(). This takes a list as a *parameter* and will `return` a number.

Example:

```
my_fav_numbers = [1, 2, 4, 7, 88]
len(my_fav_numbers) #This is 5.
```

Strings

A string is just a list of characters (e.g. alphabetical letters). Setting a variable equal to a word or a DNA sequence, for example, will create a string. Use single quotes when using strings to prevent the computer from thinking your string is referring to a variable.

Example:

```
my_favorite_word = 'door'
```

Since a string is just a list of characters, you can access individual characters the same way that you would access items in any other list. Example:

```
my_favorite_word[0] #this is 'd'
my_favorite_word[1] #this is 'o'
my_favorite_word[2] #this is 'o'
my_favorite_word[3] #this is 'r'
```

True or False?

Booleans

Sometimes you need to know if something is true or false so you can have your program make a decision. You can store the value of True or False in variables the same way that you store numbers in a variable. This is called a Boolean.

Example:

```
matt_owns_a_cat = False
matt_likes_cats = True
```

Boolean expressions

Instead of resulting in a number or something, certain expressions will `return` a Boolean value (i.e. `True` or `False`). Most often, these are expressions that are comparing values. This includes following operators:

Operator	Meaning
`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

Example:

```
a = 4
a == 5 #is False because a is 4 and 4 is not equal to 5.
(a + 1) == 5 #is True
true_or_false = ((a + 1) == 5)
#true_or_false contains the value True
```

Control Structures

If or else

Sometimes you need a program to run specific code if something (a condition) is true or false. This is possible using control structures. The most basic control structure is the `if` statement, which checks to see if an expression is True and then runs code if it is.

This uses the following syntax (i.e. format):

```
if (expression): #checks if expression is equal to True
    #if expression is True, run any code here
    #remember to indent any code that should be considered "inside" the if statement
```

Example:

```
harry = 0 #setting up variables that we will use
ron = 0
hagrid = 0
if (True): #this will always run
    harry = 1

if (False): #this will never run
    ron = 2

#end result: harry is 1, ron is 0

if (harry > ron): # 1 > 0, so this will run
    hagrid = 3
#now hagrid is 3
```

Previously, we've only been running code if conditions were true. Sometimes you might want to run different code if your condition is false. You can do this by extending your `if` statement into an `if-else`. This follows the following syntax:

```
if (condition):
    do something

else:
    do something different
```

Example:

```
harry_num_owls = 1
ron_money = 0

#ron is sad if harry has more owls than ron has money
if (harry_num_owls > ron_money): # 1 > 0 so this runs
    ron_emotional_state = 'sad' #and ron is sad
else:
    ron_emotional_state = 'happy'
```

Loops

Many times in your life, you will need to do the same thing over and over again. Luckily, you can automate this in your computer programs using loops.

`for` loop

A `for` loop repeats over all of the items in list. You can define a variable in a while loop that takes the value of the current item in the list.

Syntax:

```
example_list = [11, 12, 13, 14]
for current_item in example_list:
    print current_item
```

The output is:

```
11
12
13
14
```

You can also use the built-in function `range` to access items in a list using a for loop. The `range` function takes two numbers as parameters, in our case: zero and the length of the list (4), and generates another list from zero to len(example_list) - 1.

```
example_list = [11, 12, 13, 14]
range(example_list) #this is [0, 1, 2, 3]
```

```
example_list = [11, 12, 13, 14]
for index in range(0, len(example_list)):
    print example_list[index]
```

This does the exact same thing as the first `for` loop above.