# Python 1

## Python Overview

Python is a scripting language. It is useful for writing medium-sized scientific coding projects. When you run a Python script, the Python program will generate byte code and interpret the byte code. This happens automatically and you don't have to worry about it. Compiled languages like C and C++ will run much faster, but are much much more complicated to program. Languages like Java (which also gets compiled into byte code) are well suited to very large collaborative programming projects, but don't run as fast as C and are more complex that Python.

Python has

- data types
- functions
- objects
- classes
- methods

**Data types** are just different types of data. Examples of data types are integer numbers and strings of letters and numbers (text). These can be stored in variables.

**Functions** do something with data, such as a calculation. Some functions are already built into Python. You can create your own functions as well.

**Objects** group various data and functions (methods) that act on that data into a convenient and coherent programming paradigm.

**Classes** are a way to encapsulate (organize) variables and functions. Objects get their variables and methods from the class they belong to.

**Methods** are just functions that belong to a class. Objects that belong to the a class can use methods from that class.

## Running Python

There are two versions of Python: Python 2 and Python 3. We will be using 3. This version fixes some of the problems with Python 2 and breaks some other things. A lot of code has already been written for Python 2 (it's older), but going forwards, all new code development uses Python 3.

### Interactive Interpreter

Python can be run one line at a time in an interactive interpreter. You can think of this as a Python shell. To launch the interpreter, type the following into your terminal window:

```
$ python3
```

Note: '$' indicates the command line prompt. Recall from Unix 1 that every computer can have a different prompt! The python interpreter has its own prompt `>>>`.

First Python Commands:

```
>>> print("Hello, PFB!")
Hello, PFB!
```

> Note: `print` is a function. Function names are followed by (), so formally, the function is `print()`

## Python Scripts are Text Files

- The same code from above is typed into a text file using a text editor.
- Python scripts are always saved in files whose names have the extension '.py' (i.e. the filename ends with '.py').
- We could call the file `hello.py`

File Contents:

```
print("Hello, PFB!")
```

## Running Python Scripts

Typing the Python command followed by the name of a script makes Python execute the script. Recall that we just saw you can run an interactive interpreter by just typing `python3` on the command line.

Execute the Python script like this (% represents the prompt)

```
% python3 hello.py
```

This produces the following result in the Terminal:

```
Hello, PFB!
```

## A quicker/better way to run python scripts

If you make your script executable, you can run it without typing `python3` first. Use `chmod` to change the permissions on the script like this

`chmod +x hello.py`

You can look back at the unix lecture to learn more about permissions.

We also need to add a line at the beginning of the script that tells the shell to run python3 to interpret the script. This line starts with `#`, so it looks like a comment to python. The `!` (exclamation mark or bang) is important as is the space between `env` and `python3`. The program `/usr/bin/env` looks for where `python3` is installed and runs the script with `python3`. The details may seem a bit complex, but you can just copy and paste this 'magic' line.

The file hello.py now looks like this

```
#!/usr/bin/env python3
print("Hello, PFB!")
```

Now you can simply type the symbol for the current directory `.` followed by a `/` and the name of the script to run it. This means run the `hello.py` script in the directory I'm currently in. Like this

```
% ./hello.py
Hello, PFB!
```

Or you can add the current directory to the unix variable PATH as we saw at the end of the unix lecture, then you can just execute the script like so

```
% hello.py
```

## Syntax

### Python Variable Names

A Python variable name is a name used to identify a variable, function, class, module, or other object. A variable name starts with a letter, `A` to `Z` or `a` to `z`, or an underscore (`_`), followed by zero or more letters, underscores, and digits (`0` to `9`).

Python does not allow punctuation characters such as `@`, `$`, and `%` within a variable name. Python is a case sensitive programming language. Thus, `seq_id` and `seq_ID` are two different variable names in Python.

### Naming conventions for Python Variable Names

- The first character is lowercase, unless it is a name of a class. Classes should begin with an uppercase characters (ex. `Seq`).
- Sometimes you'll see variable names starting with one or two underscores in python libraries. We are not going to discuss these further.
- Python language-defined special names begin and end with two underscores (ex. `__file__` which is the name of the current python file).

Picking good variable names for the objects you name yourself is very important. Don't call your variables things like `items` or `my_list` or `data` or `var`. Except for where you have a very simple piece of code, or you are plotting a graph, don't call your objects `x` or `y` either. All these name examples are not descriptive of what kind of data you will find in the variable or object. Worse is to call a variable that contains gene names as `sequences`. Why is this such a bad idea? In computer science, names should always accurately describe the object they are attached to. This reduces possibility of bugs in your code, makes it much easier to understand if you come back to it after six months or share your code with someone, and makes it faster to write code that works right. Even though it takes a bit of time and effort to think up a good name for an object, it will prevent so many problems in the future!

## Reserved Words

The following is a list of Python keywords. These are special words that already have a purpose in python and therefore should not be used as variable names. If you do by accident, it overwrites the python function and causes problems.

```
and         exec        not      dict
as          finally     or
assert      for         pass
break       from        print
class       global      raise
continue    if          return
def         import      try
del         in          while
elif        is          with
else        lambda      yield
except      list        hash
```

## Lines and Indentation

Python denotes a block of code by lines with the same level of indentation. This keeps lines of code that run together organized. Incorrect line spacing and/or indention will cause an error or can make your code run in a way you don't expect. You can get help with indentation from good text editors or Interactive Development Environments (IDEs).

The number of spaces in the indentation needs to be consistent, but a specific number is not required. All lines of code, or statements, within a single block must be indented the same amount. For example, using four spaces:

```python
#!/usr/bin/env python3
message = '' # make an empty variable
for x in [1, 2, 3, 4, 5]:
    if x > 4:
        print("Hello")
        message = 'x is big'
    else:
        print(x)
        message = 'x is small'
    print(message)
print('All Done!')
```

### Comments

Including comments in your code is an essential programming practice. Making a note of what a line or block of code is doing will help the writer and readers of the code. This includes you!

Comments start with a pound or hash symbol `#`. All characters after this symbol, up to the end of the line are part of the comment and are ignored by Python.

The first line of a script starting with `#!` is a special example of a comment that also has the special function in Unix of telling the Unix shell how to run the script.

```python
#!/usr/bin/env python3

# this is my first script
print("Hello, PFB!") # this line prints output to the screen
```

### Blank Lines

Blank lines are also important for increasing the readability of the code. You should separate pieces of code that go together with a blank line to make 'paragraphs' of code. Blank lines are ignored by the Python interpretor.

## Data Types and Variables

This is our first look at variables and data types. Each data type will be discussed in more detail in subsequent sections.

The first concept to consider is that Python data types are either immutable (unchangeable) or not. Literal numbers, strings, and tuples cannot be changed. Lists, dictionaries, and sets can be changed. So can individual (scalar) variables.

You can store data in memory by putting it in different kinds of variables. You use the `=` sign to assign a value to a variable.

### Numbers and Strings

Numbers and strings are two common data types. Literal numbers and strings like this `5` or `'my name is'` are immutable. However, their values can be stored in variables, which can be changed.

For Example:

```
gene_count = 5
# change the value of gene_count
gene_count = 10
```

> Recall the section above on variable and object names (and variables are objects in Python).

Different types of data can be assigned to variables, i.e., integers (`1`,`2`,`3`), floats (floating point numbers, `3.1415`), and strings (`"text"` ).

For Example:

```
count   = 10     # this is an integer
average = 2.531    # this is a float
message = "Welcome to Python" # this is a string, contains spaces
```

`10`, `2.531`, and `"Welcome to Python"` are singular (scalar) pieces of data, and each is stored in its own variable.

Collections of data can also be stored in special data types, i.e., tuples, lists, sets, and dictionaries. You should always try to store like with like, so each element in the collection should be the same kind of data, like an expression value from RNA-seq or a count of how many exons are in a gene or a read sequence. Why do you think this might be?

## Lists

- Lists are used to store an ordered, *indexed* collection of data.
- Lists are mutable: the number of elements in the list and what's stored in each element can change
- Lists are enclosed in square brackets and items are separated by commas

```
[ 'atg' , 'aaa' , 'agg' ]
```

| Index | Value |
|---|---|
| 0 | atg |
| 1 | aaa |
| 2 | agg |

> The list index starts at 0

## Tuples

- Tuples are similar to lists and contain ordered, *indexed* collections of data.
- **Tuples are immutable: you can't change the values or the number of values**
- A tuple is enclosed in parentheses and items are separated by commas.

```
( 'Jan' , 'Feb' , 'Mar' , 'Apr' , 'May' , 'Jun' , 'Jul' , 'Aug' , 'Sep' , 'Oct' , 'Nov' ,
'Dec' )
```

| Index | Value |
|---|---|
| 0 | Jan |
| 1 | Feb |
| 2 | Mar |
| 3 | Apr |
| 4 | May |
| 5 | Jun |
| 6 | Jul |
| 7 | Aug |
| 8 | Sep |
| 9 | Oct |
| 10 | Nov |
| 11 | Dec |

## Dictionaries

- Dictionaries are good for storing data that can be represented as a two-column table.
- They store ordered collections of key/value pairs. (In version 3.6 and earlier, they were not ordered)
- A dictionary is enclosed in curly braces.
- A colon is written between each key and value. Commas separate key:value pairs.

```
{
  'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' ,
  'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
}
```

| Key | Value |
|---|---|
| TP53 | GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC |
| BRCA1 | GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA |

## Command line parameters: A Special Built-in List

Command line parameters follow the name of a script or program and have spaces between them. They allow a user to pass information to a script on the command line when that script is being run. Python stores all the pieces of the command line in a special list called `sys.argv`.

You need to import the module named `sys` at the beginning of your script like this

```
#!/usr/bin/env python3
import sys
```

Let's imagine we have a script called 'friends.py'. If you write this on the command line:

```
$ friends.py Joe Anita
```

This happens inside the script ('friends.py'):

> the strings 'Joe' and 'Anita' appear in a list called `sys.argv`.

> These are the command line parameters, or arguments you want to pass to your script.
> `sys.argv[0]` is the script name.
> You can access values of the other parameters by their indices, starting with 1, so `sys.argv[1]` contains 'Joe' and `sys.argv[2]` contains 'Anita'. You access elements in a list by adding square brackets and the numerical index after the name of the list.
> If you wanted to print a message saying these two people are friends, you might write some code like this

```
#!/usr/bin/env python3
import sys

friend1 = sys.argv[1] # get first command line parameter
friend2 = sys.argv[2] # get second command line parameter

# Now print a message to the screen
print(friend1, 'and', friend2, 'are friends')
```

The advantage of getting input from the user from the command line is that you can write a script that is general. It can print a message with any input the user provides. This makes it flexible.
The user also supplies all the data the script needs on the command line so the script doesn't have to ask the user to input a name and wait until the user does this. The script can run on its own with no further interaction from the user. This frees the user to work on something else. Very handy!

## What kind of object am I working with?

You have an identifier in your code called `data`. Does it represent a string or a list or a dictionary? Python has a couple of functions that help you figure this out.

| Function | Description |
|---|---|
| `type(data)` | tells you which class your object belongs to |
| `dir(data)` | tells you which methods are available for your object |
| `id(data)` | tells you the unique object id |

We'll cover `dir()` in more detail later

```
>>> data = [2, 4, 6]
>>> type(data)
<class 'list'>
>>> data = 5
>>> type(data)
<class 'int'>
>>> id(data)
44990666544
```

---

## Link to Python 1 Problem Set

---

# Python 2

## Operators

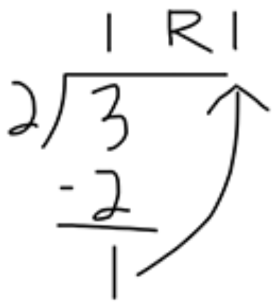
An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce a result. Here we explain the concept of operators.

### Arithmetic Operators

In Python we can write statements that perform mathematical calculations. To do this we need to use operators that are specific for this purpose. Here are arithmetic operators:

| Operator | Description | Example | Result |
|---|---|---|---|
| `+` | Addition | `3+2` | `5` |
| `-` | Subtraction | `3-2` | `1` |
| `*` | Multiplication | `3*2` | `6` |
| `/` | Division | `3/2` | `1.5` |
| `%` | Modulus (divides left operand by right operand and returns the remainder) | `3%2` | `1` |
| `**` | Exponent | `3**2` | `9` |
| `//` | Floor Division (result is the quotient with digits after the decimal point removed. If one of the operands is negative, the result is rounded to the integer further away from zero | `3//2` `-3//2` | `1` `-2` |

**Modulus**



## Assignment Operators

We use assignment operators to assign values to variables. You have been using the `=` assignment operator. Here are others:

| Operator | Equivalent to | Example | result evaluates to |
|---|---|---|---|
| `=` | `result = 3` | `result = 3` | 3 |
| `+=` | `result = result + 2` | `result = 3; result += 2` | 5 |
| `-=` | `result = result - 2` | `result = 3; result -= 2` | 1 |
| `*=` | `result = result * 2` | `result = 3; result *= 2` | 6 |
| `/=` | `result = result / 2` | `result = 3; result /= 2` | 1.5 |
| `%=` | `result = result % 2` | `result = 3; result %= 2` | 1 |
| `**=` | `result = result ** 2` | `result = 3; result **= 2` | 9 |
| `//=` | `result = result // 2` | `result = 3; result //= 3` | 1 |

## Comparison Operators

These operators compare two values and returns true or false.

| Operator | Description | Example | Result |
|---|---|---|---|
| `==` | equal to | `3 == 2` | False |
| `!=` | not equal | `3 != 2` | True |
| `>` | greater than | `3 > 2` | True |
| `<` | less than | `3 < 2` | False |
| `>=` | greater than or equal | `3 >= 2` | True |
| `<=` | less than or equal | `3 <= 2` | False |

## Logical Operators

Logical operators allow you to combine two or more sets of comparisons. You can combine the results in different ways. For example you can 1) demand that all the statements are true, 2) that only one statement needs to be true, or 3) that the statement needs to be false.

| Operator | Description | Example | Result |
|---|---|---|---|
| `and` | True if left operand is True and right operand is True | `3>=2 and 2<3` | True |
| `or` | True if left operand is True or right operand is True | `3==2 or 2<3` | True |
| `not` | Reverses the logical status | `not False` | True |

## Membership Operators

You can test to see if a value is included in a string, tuple, or list. You can also test that the value is not included in the string, tuple, or list.

| Operator | Description |
|---|---|
| `in` | True if a value is included in a list, tuple, or string |
| `not in` | True if a value is absent in a list, tuple, or string |

For Example:

```
>>> dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> 'TCT' in dna
True
>>>
>>> 'ATG' in dna
False
>>> 'ATG' not in dna
True
>>> codons = ['atg' , 'aaa' , 'agg']
>>> 'atg' in codons
True
>>> 'ttt' in codons
False
```

## Operator Precedence

Operators are listed in order of precedence. Highest listed first. Not all the operators listed here are mentioned above.

| Operator | Description |
|---|---|
| `**` | Exponentiation (raise to the power) |
| `~` `+` `-` | Complement/Bitwise inverse, unary plus and minus (method names for the last two are +@ and -@) |
| `*` `/` `%` `//` | Multiply, divide, modulo and floor division |
| `+` `-` | Addition and subtraction |
| `>>` `<<` | Right and left bitwise shift |
| `&` | Bitwise 'AND' |
| `^` `\|` | Bitwise exclusive 'OR' and regular 'OR' |
| `<=` `<` `>` `>=` | Comparison operators |
| `<>` `==` `!=` | Equality operators |
| `=` `%=` `/=` `//=` `-=` `+=` `*=` `**=` | Assignment operators |
| `is` | Identity operator |
| `is not` | Non-identity operator |
| `in` | Membership operator |
| `not in` | Negative membership operator |
| `not` `or` `and` | logical operators |

Note: Find out more about bitwise operators.

## Truth

Lets take a step back... What is truth?

Everything is true, except for:

| expression | TRUE/FALSE |
|---|---|
| `0` | FALSE |
| `None` | FALSE |
| `False` | FALSE |
| `''` (empty string) | FALSE |
| `[]` (empty list) | FALSE |
| `()` (empty tuple) | FALSE |
| `{}` (empty dictionary) | FALSE |

Which means that these are True:

| expression | TRUE/FALSE |
|---|---|
| `'0'` | TRUE |
| `'None'` | TRUE |
| `'False'` | TRUE |
| `'True'` | TRUE |
| `' '` (string of one blank space) | TRUE |

## Use `bool()` to test for truth

`bool()` is a function that will test if a value is true.

```
>>> bool(True)
True
>>> bool('True')
True
>>>
>>>
>>> bool(False)
False
>>> bool('False')
True
>>>
>>>
>>> bool(0)
```

```
False
>>> bool('0')
True
>>>
>>>
>>> bool('')
False
>>> bool(' ')
True
>>>
>>>
>>> bool(())
False
>>> bool([])
False
>>> bool({})
False
```

## Logic: Control Statements

Control Statements are used to direct the flow of your code and create the opportunity for decision making. The foundation of control statements is building on truth.

### If Statement

- Use the `if` Statement to test for truth and to execute lines of code if true.
- When the expression evaluates to true each of the statements indented below the `if` statement, also known as a *block*, will be executed.

**if**

```
if expression :
  statement
  statement
```

For Example:

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'AGC' in dna:
  print('Found AGC in your dna sequence')
```

Returns:

```
found AGC in your dna sequence
```

**else**

- The `if` portion of the if/else statement behaves as before.
- The first indented block is executed if the condition is true.
- If the condition is false, the second indented else block is executed.

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'ATG' in dna:
  print('Found ATG in your dna sequence')
else:
  print('Did not find ATG in your dna sequence')
```

Returns:

```
Did not find ATG in your dna sequence
```

## if/elif

- The `if` condition is tested as before, and the indented block is executed if the condition is true.
- If it's false, the indented block following the `elif` is executed if the first `elif` condition is true.
- Any remaining `elif` conditions will be tested in order until one is found to be true. If none is true, the `else` indented block is executed.

```
count = 60
if count < 0:
  print(count, "is less than 0")
elif count < 50:
  print(count, "is less than 50")
elif count > 50:
  print(count, "is greater than 50")
else:
  print(count, "must be 50")
```

Returns:

```
60 is greater than 50
```

Let's change count to 20, which statement block gets executed?

```python
count = 20
if count < 0:
  print(count, "is less than 0")
elif count < 50:
  print(count, "is less than 50")
elif count > 50:
  print(count, "is greater than 50")
else:
  print(count, "must be 50")
```

Returns:

```
20 is less than 50
```

What happens when count is 50?

```python
count = 50
if count < 0:
  print(count, "is less than 0")
elif count < 50:
  print(count, "is less than 50")
elif count > 50:
  print(count, "is greater than 50")
else:
  print(count, "must be 50")
```

Returns:

```
50 must be 50
```

## Numbers

Python recognizes 3 types of numbers: integers, floating point numbers, and complex numbers.

### integer

- known as an `int`
- an `int` can be positive or negative
- and **does not** contain a decimal point or exponent.

### floating point number

- known as a `float`
- a floating point number can be positive or negative
- and **does** contain a decimal point (`4.875`) or exponent (`4.2e-12`)

## complex number

- known as `complex`
- is in the form of a+bi where bi is the imaginary part.

## Conversion functions

Sometimes one type of number needs to be changed to another for a function to be able to do work on it. Here are a list of functions for converting number types:

| function | Description |
|---|---|
| `int(x)` | to convert x to a plain integer |
| `float(x)` | to convert x to a floating-point number |
| `complex(x)` | to convert x to a complex number with real part x and imaginary part zero |
| `complex(x, y)` | to convert x and y to a complex number with real part x and imaginary part y |

```
>>> int(2.3)
2
>>> float(2)
2.0
>>> complex(2.3)
(2.3+0j)
>>> complex(2.3,2)
(2.3+2j)
```

## Numeric Functions

Here is a list of functions that take numbers as arguments. These do useful things like rounding.

| function | Description |
|---|---|
| `abs(x)` | The absolute value of x: the (positive) distance between x and zero. |
| `round(x [,n])` | x rounded to n digits from the decimal point. round() rounds to an even integer if the value is exactly between two integers, so round(0.5) is 0 and round(-0.5) is 0. round(1.5) is 2 |
| `max(x1, x2,...)` | The largest argument is returned |
| `min(x1, x2,...)` | The smallest argument is returned |

**Rounding to a fixed number of decimal places can give unpredictable results.**

> Python's round() function for floats uses a rounding strategy known as "round half to even" (also called "banker's rounding") when a number is exactly equidistant between two possible rounded values (e.g., 2.5, 3.5).

> In such cases, it rounds to the nearest even number. So, round(2.5) is 2, and round(3.5) is 4.

> This rule, combined with the floating-point imprecision, can lead to unexpected outcomes. If 2.675 is internally represented as 2.67499999999999999, round(2.675, 2) will round down to 2.67, even though one might intuitively expect 2.68.

```
>>> abs(2.3)
2.3
>>> abs(-2.9)
2.9
>>> round(2.3)
2
>>> round(2.5)
2
>>> round(2.9)
3
>>> round(-2.9)
-3
>>> round(-2.3)
-2
>>> round(-2.009,2)
-2.01
>>> round(2.675, 2)  # note this rounds down
2.67
>>> max(4,-5,5,1,11)
11
>>> min(4,-5,5,1,11)
-5
```

Many numeric functions are not built into the Python core and need to be imported into our program if we want to use them. To include them, add the following to the top of the program:
`import math`

The following functions are found in the `math` module and must be imported. To use these functions, prepend the function with the module name, i.e, `math.ceil(15.5)`

| math.function | Description |
|---|---|
| `math.ceil(x)` | return the smallest integer greater than or equal to x is returned |
| `math.floor(x)` | return the largest integer less than or equal to x. |
| `math.exp(x)` | The exponential of x: $e^x$ is returned |
| `math.log(x)` | the natural logarithm of x, for x > 0 is returned |
| `math.log10(x)` | The base-10 logarithm of x for x > 0 is returned |
| `math.modf(x)` | The fractional and integer parts of x are returned in a two-item tuple. |
| `math.pow(x, y)` | The value of x raised to the power y is returned |
| `math.sqrt(x)` | Return the square root of x for x >= 0 |

```
>>> import math
>>>
>>> math.ceil(2.3)
3
>>> math.ceil(2.9)
3
>>> math.ceil(-2.9)
-2
>>> math.floor(2.3)
2
>>> math.floor(2.9)
2
>>> math.floor(-2.9)
-3
>>> math.exp(2.3)
9.974182454814718
>>> math.exp(2.9)
18.17414536944306
>>> math.exp(-2.9)
0.05502322005640723
>>>
>>> math.log(2.3)
0.8329091229351039
```

```
>>> math.log(2.9)
1.0647107369924282
>>> math.log(-2.9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
>>> math.log10(2.3)
0.36172783601759284
>>> math.log10(2.9)
0.4623979978989561
>>>
>>> math.modf(2.3)
(0.2999999999999998, 2.0)
>>>
>>> math.pow(2.3,1)
2.3
>>> math.pow(2.3,2)
5.289999999999999
>>> math.pow(-2.3,2)
5.289999999999999
>>> math.pow(2.3,-2)
0.18903591682419663
>>>
>>> math.sqrt(25)
5.0
>>> math.sqrt(2.3)
1.51657508881031
>>> math.sqrt(2.9)
1.70293863659264
```

## Comparing two numbers

Oftentimes, it is necessary to compare two numbers and find out if the first number is less than, equal to, or greater than the second.

The simple function `cmp(x,y)` is not available in Python 3.

Use this idiom instead:

```
cmp = (x>y)-(x<y)
```

It returns three different values depending on x and y

- if x<y, then -1 is returned
- if x>y, then 1 is returned
- x == y, then 0 is returned

## Link to Python 2 Problem Set