# Lecture3

August 21, 2018

## 1 Lecture 3: Basics of Python

CBIO (CSCI) 4835/6835: Introduction to Computational Biology

### 1.1 Overview and Objectives

In this lecture, I'll introduce the Python programming language and how to interact with it; aka, the proverbial Hello, World! lecture. By the end, you should be able to:

- Recall basic history and facts about Python (relevance in scientific computing, comparison to other languages)
- Print arbitrary strings in a Python environment
- Create and execute basic arithmetic operations
- Understand and be able to use variable assignment and update
- Define variables of string and numerical types, convert between them, and use them in basic operations
- Explain the different variants of typing in programming languages, and what "duck-typing" in Python does
- Understand how Python uses whitespace in its syntax
- Demonstrate how smart variable-naming and proper use of comments can effectively document your code

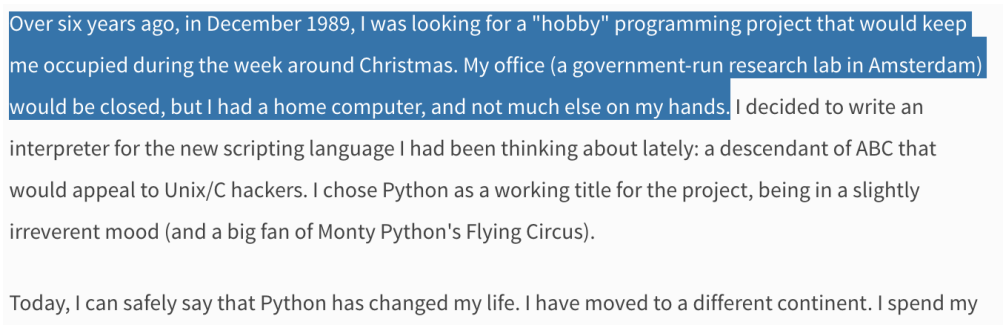
### 1.2 Part 1: Background

Python as a language was implemented from the start by Guido van Rossum. What was originally something of a snarkily-named hobby project to pass the holidays turned into a huge open source phenomenon used by millions.

#### 1.2.1 Python's history

The original project began in 1989.

- Release of Python 2.0 in 2000
- Release of Python 3.0 in 2008
- Latest stable release of these branches are **2.7.15**–which Guido *emphatically* insists is the final, final, final release of the 2.x branch–and **3.6** (which is what we're using in this course)




guido

Wondering why a 2.x branch has survived *almost two decades* after its initial release?

Python 3 was designed as backwards-incompatible; a good number of syntax changes and other internal improvements made the majority of code written for Python 2 unusable in Python 3.

This made it difficult for power users and developers to upgrade, particularly when they relied on so many third-party libraries for much of the heavy-lifting in Python.

Until these third-party libraries were themselves converted to Python 3 (really only in the past couple years!), most developers stuck with Python 2.

### 1.2.2 Python, the Language

Python is an **intepreted** language.

- Contrast with **compiled** languages
- Performance, ease-of-use
- Modern intertwining and blurring of compiled vs interpreted languages

Python is a very **general** language.

- Not designed as a specialized language for performing a specific task. Instead, it relies on third-party developers to provide these extras.

Instead, as Jake VanderPlas put it:

> "Python syntax is the glue that holds your data science code together. As many scientists and statisticians have found, Python excels in that role because it is powerful, intuitive, quick to write, fun to use, and above all extremely useful in day-to-day data science tasks."

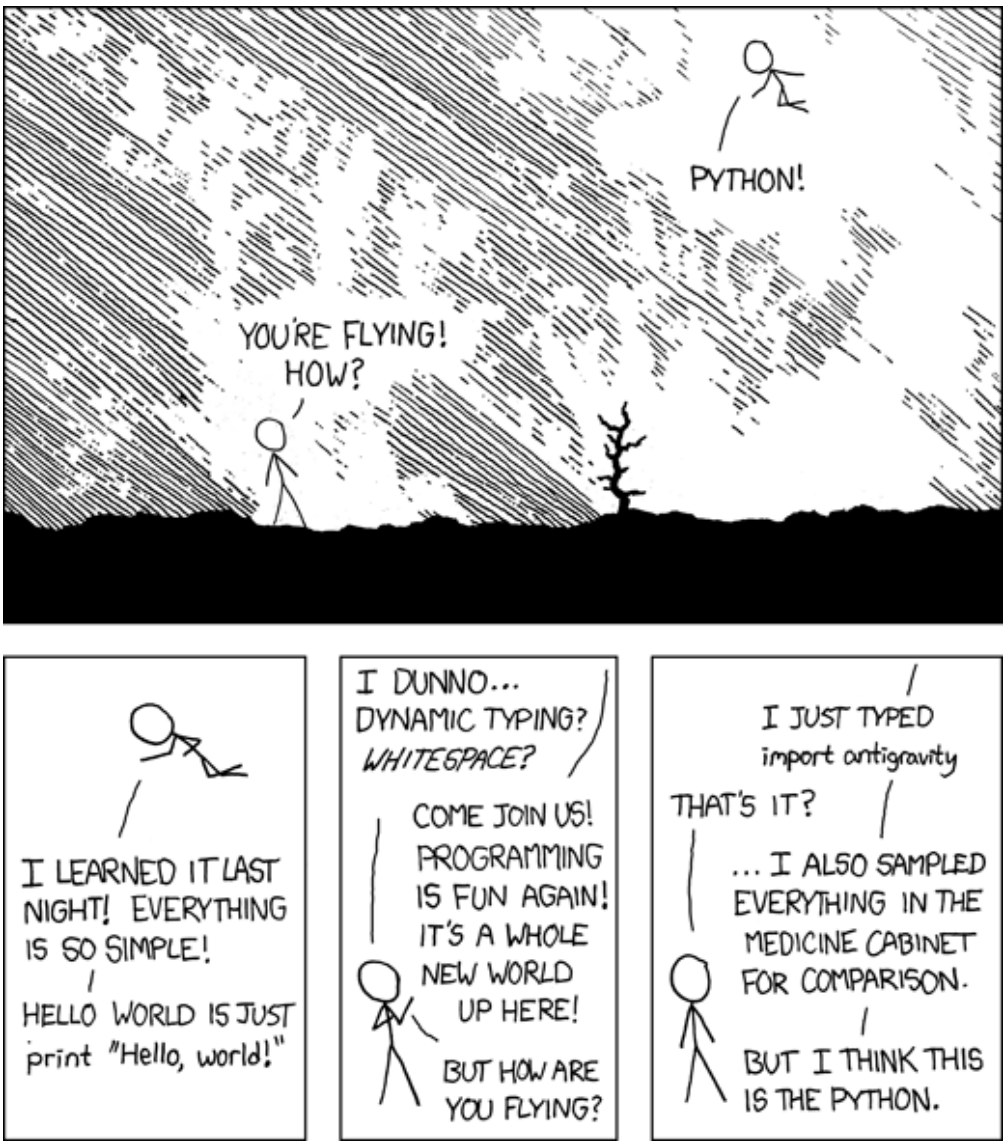
### 1.2.3 Zen of Python

One of the biggest reasons for Python's popularity is its overall simplicity and ease of use.

Python was designed *explicitly* with this in mind!

It's so central to the Python ethos, in fact, that it's baked into every Python installation. Tim Peters wrote a "poem" of sorts, *The Zen of Python*, that anyone with Python installed can read.

To see it, just type one line of Python code:


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

xkcd

```
In [1]: import this
```

```
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!
```

Lack of any discernible meter or rhyming scheme aside, it nonetheless encapsulates the spirit of the Python language. These two lines are particular favorites of mine:

Line 1: - If you wrote the code and can't explain it*, go back and fix it. - If you didn't write the code and can't explain it, get the person who wrote it to fix it.

Line 2: - "Easy to explain": necessary and sufficient for good code?

Don't you just feel so zen right now?

### 1.3 Part 2: Hello, World!

Enough reading, time for some coding, amirite?

So what does it take to write your first program, your "Hello, World!"? Pound-include iostream dot h? Import java dot io? Define a main function with command-line parameters? Wrap the whole thing in a class?

```
In [2]: print("Hello, world!")
```

```
Hello, world!
```

Yep! That's all there is to it.

Just for the sake of being thorough, though, let's go through this command in painstaking detail.

**Functions**: `print()` is a function.

- Functions take input, perform an operation on it, and give back (return) output.

You can think of it as a direct analog of the mathematical term, $f(x) = y$. In this case, $f$ is the function; $x$ is the input, and $y$ is the output.
Later in the course, we'll see how to create our own functions, but for now we'll make use of the ones Python provides us by default.

**Arguments**: the input to the function.

- Interchangeable with "parameters".

In this case, there is only one argument to `print()`: a string of text that we want printed out. This text, in Python parlance, is called a "string". I can only presume it is so named because it is a *string* of individual characters.

We can very easily change the argument we pass to `print()`:

```
In [3]: print("This is not the same argument as before.")
```

```
This is not the same argument as before.
```

We could also print out an empty string, or even no string at all.

```
In [4]: print("")  # this is an empty string
```

```
In [5]: print()  # this is just nothing
```

In both cases, the output looks pretty much the same…because it is: just a blank line.

- After `print()` finishes printing your input, it prints one final character–a *newline*.

This is basically the programmatic equivalent of hitting `Enter` at the end of a line, moving the cursor down to the start of the next line.

### 1.3.1 What are "strings"?

Briefly–a type of data format in Python that exclusively uses alphanumeric (A through Z, 0 through 9) characters.

Look for the double-quotes (or even single-quotes; unlike the command-line, they do the same thing in Python):

```
In [6]: "5"  # This is a string.
        '5'  # This is also a string.
        5    # This is NOT a string.
```

```
Out[6]: 5
```

### 1.3.2 What are the hashtags? (#)

They indicate *comments*.

- Comments are lines in your program that the language ignores entirely.
- When you type a # in Python, everything *after* that symbol on the same line is ignored.

They're there purely for the coders as a way to put documentation and clarifying statements directly into the code. It's a practice I **strongly** encourage everyone to do–even just to remind yourself what you were thinking! I can't count the number of times I worked on code, set it aside for a month, then came back to it and had absolutely no idea what I was doing)

## 1.4 Part 3: Beyond "Hello, World!"

Ok, so Python can print strings. That's cool. Can it do anything that's actually useful?

Python has a lot of built-in objects and data structures that are very useful for more advanced operations–and we'll get to them soon enough!–but for now, you can use Python to perform basic arithmetic operations.

Addition, subtraction, multiplication, division–they're all there. You can use it as a glorified calculator:

```
In [7]: 3 + 4

Out[7]: 7

In [8]: 3 - 4

Out[8]: -1

In [9]: 3 * 4

Out[9]: 12

In [10]: 3 / 4

Out[10]: 0.75
```

Python respects order of operations, too, performing them as you'd expect:

```
In [11]: 3 + 4 * 6 / 2 - 5

Out[11]: 10.0

In [12]: (3 + 4) * 6 / (2 - 5)

Out[12]: -14.0
```

Python even has a really cool exponent operator, denoted by using two stars right next to each other:

```
In [13]: 2 ** 3  # 2 raised to the 3rd power
```

```
Out[13]: 8

In [14]: 3 ** 2  # 3 squared

Out[14]: 9

In [15]: 25 ** (1 / 2)  # Square root of 25

Out[15]: 5.0
```

Now for something really neat:

```
In [16]: x = 2
         x * 3

Out[16]: 6
```

This is an example of using Python *variables*.

- Variables store and maintain values that can be updated and manipulated as the program runes.
- You can name a variable whatever you like, as long as it doesn't start with a number ("`5var`" would be illegal, but "`var5`" would be fine) or conflict with reserved Python words (like `print`).

Here's an operation that involves two variables:

```
In [17]: x = 2
         y = 3
         x * y

Out[17]: 6
```

We can assign the result of operations with variables to other variables:

```
In [18]: x = 2
         y = 3
         z = x * y
         print(z)

6
```

The use of the equals sign = is called the *assignment operator*.

- "Assignment" takes whatever value is being computed on the *right-hand* side of the equation and **assigns** it to the variable on the *left-hand* side.
- Multiplication (*), Division (/), Addition (+), and Subtraction (-) are also *operators*.

What happens if I perform an assignment on something that can't be assigned a different value…such as, say, a number?

```
In [19]: x = 2
         y = 3
```

**CRASH!**

Ok, not really; Python technically did what it was supposed to do. It threw an error, alerting you that something in your program didn't work for some reason. In this case, the error message is `can't assign to literal`.

Parsing out the `SyntaxError` message:

- `Error` is an obvious hint. `Syntax` gives us some context.
- We did something wrong that involves Python's syntax, or the structure of its language.

The "`literal`" being referred to is the number 5 in the statement: `5 = x * y`

- We are attempting to assign the result of the computation of `x * y` to the number 5
- However, 5 is known internally to Python as a "literal"
- 5 is literally 5; you can't change the value of 5! (5 = 8? NOPE)

So we can't assign values to numbers. What about assigning values to a variable that's used in the very same calculation?

```
In [20]: x = 2
         y = 3
         x = x * y

In [21]: print(x)

6
```

This works just fine! In fact, it's more than fine–this is such a standard operation, it has its own operator:

```
In [22]: x = 2
         y = 3
         x *= y
         print(x)

6
```

Out loud, it's pretty much what it sounds like: "x times equals y". Put another way, you're *updating* or *reassigning* an existing variable with a new value. **This happens A LOT.**

This is an instance of a shorthand operator.

- We multiplied `x` by `y` and stored the product in `x`, effectively updating it.
- There are many instances where you'll want to increment a variable: for example, when counting how many of some "thing" you have.
- All the other operators have the same shorthand-update versions: `+=` for addition, `-=` for subtraction, and `/=` for division.

## 1.5 Part 4: Variables and Types

We've seen how we can create variables to assign, update, and combine values using specific operators. It's important to note that all variables have **types** that will dictate much (if not all) of the operations you can perform on and with that variable.

There are two critical components to every single variable you will ever create in Python: the variable's *name* and its *type*.

```
In [23]: x = 2
```

It's easy to determine the name of the variable; in this case, the name is $x$. It can be a bit more complicated to determine the type of the variable, as it depends on the value the variable is storing. In this case, it's storing the number 2. Since there's no decimal point on the number, we call this number an *integer*, or *int* for short.

### 1.5.1 Numerical types

What other types of variables are there?

```
In [24]: y = 2.0
```

`y` is assigned a value of 2.0: it is referred to as a *floating-point* variable, or *float* for short. Any number with a decimal (i.e. fractional number) is considered a float.

Floats do the heavy-lifting of much of the computation in data science. Whenever you're computing probabilities or fractions or normalizations, floats are the types of variables you're using. In general, you tend to use floats for heavy computation, and ints for counting things.

There is an explicit connection between ints and floats. Let's illustrate with an example:

```
In [25]: x = 2
         y = 3
         z = x / y
```

In this case, we've defined two variables `x` and `y` and assigned them integer values, so they are both of type `int`. However, we've used them both in a division operation and assigned the result to a variable named `z`. If we were to check the type of `z`, what type do you think it would be?

```
In [26]: type(z)

Out[26]: float
```

**Why?**

In general, an operation involving two things of one type will give you a result that's the same type. However, in cases where a decimal number is outputted, Python implicitly "promotes" the variable storing the result.

Changing the type of a variable is known as **casting**, and it can take two forms: implicit casting (as we just saw), or explicit casting, where you (the programmer) tell Python to change the type of a variable.

### 1.5.2 Casting

Implicit casting is done in such a way as to try to abide by "common sense": if you're dividing two numbers, you would all but expect to receive a fraction, or decimal, on the other end. If you're multiplying two numbers, the type of the output depends on the types of the inputs–two floats multiplied will likely produce a float, while two ints multiplied will produce an int.

```
In [27]: x = 2
         y = 3
         z = x * y
         type(z)

Out[27]: int

In [28]: x = 2.5
         y = 3.5
         z = x * y
         type(z)

Out[28]: float
```

In both cases, the type of the inputs dictated the type of the outputs.

Explicit casting, on the other hand, is a little trickier. In this case, it's *you the programmer* who are making explicit (hence the name) what type you want your variables to be.

Python has a couple special built-in functions for performing explicit casting on variables, and they're named what you would expect: `int()` for casting a variable as an int, and `float()` for casting it as a float.

```
In [29]: x = 2.5
         y = 3.5
         z = x * y

In [30]: print(z)  # What type is this?

8.75


In [31]: print(int(z))  # Now what type is this?

8
```

With explicit casting, you are telling Python to override its default behavior. In doing so, it has to make some decisions as to how to do so in a way that still makes sense.

When you cast a `float` to an `int`, some information is lost; namely, the decimal. So the way Python handles this is by quite literally **discarding the entire decimal portion**.

In this way, even if your number was 9.999999999 and you perfomed an explicit cast to `int()`, Python would hand you back a 9.

### 1.5.3 Language typing mechanisms

Python as a language is known as *dynamically typed*. This means you don't have to specify the type of the variable when you define it; rather, Python infers the type based on how you've defined it and how you use it.

As we've already seen, Python creates a variable of type `int` when you assign it an integer number like 5, and it automatically converts the type to a `float` whenever the operations produce decimals.

Other languages, like C++ and Java, are *statically typed*, meaning in addition to naming a variable when it is declared, the programmer must also explicitly state the type of the variable.

Pros and cons of *dynamic* typing (as opposed to *static* typing)?

Pros: - Streamlined - Flexible

Cons: - Easier to make mistakes - Potential for malicious bugs

All languages have some form of *type-checking*: that is, ensuring that the types of the variables are compatible with the operations you're performing on them.

After all–you can't multiply strings together, right? What would the output of `"this" * "that"` be? Nothing that makes any sense, surely. So [most] languages simply don't allow it, by way of type-checking.

For type-checking, Python implements what is known as duck typing: if it walks like a duck and quacks like a duck, it's a duck.

This brings us to a concept known as **type safety**. A particularly fun example is known as a roundoff error, or more specifically to our case, a representation error. This occurs when we are attempting to represent a value for which we simply don't have enough precision to accurately store.

- When there are too many decimal values to represent (usually because the number we're trying to store is very, very small), we get an *underflow error*.
- When there are too many whole numbers to represent (usually because the number we're trying to store is very, very large), we get an *overflow error*.

One of the most popular examples of an overflow error was the Y2K bug. In this case, most Windows machines internally stored the year as simply the last two digits. Thus, when the year 2000 rolled around, the two numbers representing the year overflowed and reset to 00. A similar problem is anticipated for 2038, when 32-bit Unix machines will also see their internal date representations overflow to 0.

In these cases, and especially in dynamically typed languages like Python, it is very important to know what types of variables you're working with and what the limitations of those types are.

### 1.5.4 String types

Strings, as we've also seen previously, are the variable types used in Python to represent text.

```
In [32]: x = "this is a string"
         type(x)

Out[32]: str
```

Unlike numerical types like ints and floats, you can't really perform arithmetic operations on strings, *with one exception*:

```
In [33]: x = "some string"
         y = "another string"
         z = x + y
         print(z)
```

```
some stringanother string
```

The + operator, when applied to strings, is called *string concatenation*.

This means that it glues or *concatenates* two strings together to create a new string. In this case, we took the string in `x` and concatenated that to the string in `y`, storing the whole thing in a final string `z`.

It's a somewhat blunt operation, but it behaves somewhat intuitively.

That said, don't try to subtract, multiply, or divide strings.

**And don't add numbers that happen to be strings together, either.** This is where *knowing the type* of your variables is very important!

Casting, however, is alive and well with strings. In particular, if you know the string you're working with is a *string representation of a number*, you can cast it from a string to a numeric type:

```
In [34]: s = "2"  # Don't accidentally add this!!! Since it's a string, it would CONCATENATE.
         print(s)
         print(type(s))
```

```
2
<class 'str'>
```

```
In [35]: x = int(s)  # Now, we'll cast the string to an integer.
         print(x)
         print(type(x))
```

```
2
<class 'int'>
```

And back again:

```
In [36]: x = 2  # Start with an integer.
         print(x)
         print(type(x))
```

```
2
<class 'int'>
```

```
In [37]: s = str(x)  # Cast it back into a string.
         print(s)
         print(type(s))
```

```
2
<class 'str'>
```

### 1.5.5 Variable Comparisons and Boolean Types

We can also compare variables! By comparing variables, we can ask whether two things are equal, or greater than or less than some other value.

This sort of true-or-false comparison gives rise to yet another type in Python: the *boolean* type. A variable of this type takes only two possible values: `True` or `False`.

This is exactly the same as doing addition `+`, subtraction `-`, multiplication `*`, or division `/`, except instead of getting back a numeric type (`int` or `float`), we get back a *boolean* type that indicates whether the comparison was `True` or `False`.

Let's say we have two numeric variables, `x` and `y`, and want to check if they're equal. To do this, we use a the *comparison operator*, the double-equals == sign:

```
In [38]: x = 2
         y = 2
         z = (x == y)  # COMPARE if x is equals to y, and store the resulting True/False in z
         print(z)
```

```
True
```

The == sign is the equality comparison operator, and it will return `True` or `False` depending on whether or not the two values are exactly equal. This works for strings as well:

```
In [39]: s1 = "a string"
         s2 = "a string"
         z = (s1 == s2)
         print(z)
```

```
True
```

```
In [40]: s3 = "another string"
         z = (s1 == s3)
         print(z)
```

```
False
```

**Be careful with this operator.** It looks a lot like the assignment operator, =. But it is very different.

- With *assignment*, you are **ordering**: put the value on the right into the variable on the left
- With *comparison*, you are **asking**: are these two things exactly equal?

In addition to asking if things are equal, we can also ask if variables are less than or greater than each other, using the < and > operators, respectively.

```
In [41]: x = 1
         y = 2
         z = (x < y)  # Just as before, the result of this operator is a True/False.
         print(z)
```

```
True
```

```
In [42]: z = (x > y)
         print(z)
```

```
False
```

In a small twist of relative magnitude comparisons, we can also ask if something is less than *or equal to* or greater than *or equal to* some other value. To do this, in addition to the comparison operators < or >, we also add an equal sign:

```
In [43]: x = 2
         y = 3
         z = (x <= y)  # Less than or equal to
         print(z)
```

```
True
```

```
In [44]: x = 3
         z = (x >= y)  # Greater than or equal to
         print(z)
```

```
True
```

```
In [45]: x = 3.00001
         z = (x <= y)  # Less than or equal to
         print(z)
```

```
False
```

Interestingly, these operators also work for strings. Be careful, though: their behavior may be somewhat unexpected until you figure out what actual trick is happening:

```
In [46]: s1 = "some string"
         s2 = "another string"
         z = (s1 > s2)
         print(z)
```

```
True
```

```
In [47]: s1 = "Some string"
         z = (s1 > s2)
         print(z)
```

```
False
```

## 1.6 Part 5: Naming Conventions and Documentation

There are some rules regarding what can and cannot be used as a variable name.

Beyond those rules, there are guidelines.

### 1.6.1 Naming Rules

- Names can contain only letters, numbers, and underscores.

All the letters a-z (upper and lowercase), the numbers 0-9, and underscores are at your disposal. Anything else is illegal. No special characters like pound signs, dollar signs, or percents are allowed. Hashtag alphanumerics only.

- Variable names can only *start* with letters or underscores.

Numbers cannot be the first character of a variable name. `message_1` is a perfectly valid variable name; however, `1_message` is not and will throw an error.

- Spaces are not allowed in variable names.

Underscores are how Python programmers tend to "simulate" spaces in variable names, but simply put there's no way to name a variable with multiple words separated by spaces.

### 1.6.2 Naming Guidelines

These are not hard-and-fast rules, but rather suggestions to help "standardize" code and make it easier to read by people who aren't necessarily familiar with the code you've written.

- Make variable names short, but descriptive.

I've been giving a lot of examples using variables named `x`, `s`, and so forth. **This is bad.** Don't do it–unless, for example, you're defining `x` and `y` to be points in a 2D coordinate axis, or as a counter; one-letter variable names for counters are quite common.

Outside of those narrow use-cases, the variable names should constitute a pithy description that reflects their function in your program. A variable storing a name, for example, could be `name` or even `student_name`, but don't go as far as to use `the_name_of_the_student`.

- Be careful with the lowercase `l` or uppercase `O`.

This is one of those annoying rules that largely only applies to one-letter variables: stay away from using letters that also bear striking resemblance to numbers. Naming your variable `l` or `O` may confuse downstream readers of your code, making them think you're sprinkling 1s and 0s throughout your code.

- Variable names should be all lowercase, using underscores for multiple words.

Java programmers may take umbrage with this point: the convention there is to `useCamelCase` for multi-word variable names.

Since Python takes quite a bit from the C language (and its back-end is implemented in C), it also borrows a lot of C conventions, one of which is to use underscores and all lowercase letters in variable names. So rather than `multiWordVariable`, we do `multi_word_variable`.

The one exception to this rule is when you define variables that are *constant*; that is, their values don't change. In this case, the variable name is usually in all-caps. For example: `PI = 3.14159`.

- Avoid using Python keywords or function names as variables.

This might take some trial-and-error. Basically, if you try to name a variable `print` or `float` or `str`, you'll run into a lot of problems down the road. *Technically* this isn't outlawed in Python, but it will cause a lot of headaches later in your program.

### 1.6.3 Self-documenting code

The practice of pithy but precise variable naming strategies is known as "self-documenting code."

We've learned before that we can insert comments into our code to explain things that might otherwise be confusing:

```
In [48]: # Adds two numbers that are initially strings by converting them to an int and a float,
         # then converting the final result to an int and storing it in the variable x.
         x = int(int("1345") + float("31.5"))
         print(x)
```

```
1376
```

Comments are important to good coding style and should be used often for clarification.

However, even more preferable to the liberal use of comments is a good variable naming convention. For instance, instead of naming a variable "x" or "y" or "c", give it a name that describes its purpose.

```
In [49]: str_length = len("some string")
```

I could've used a comment to explain how this variable was storing the length of the string, but by naming the variable itself in terms of what it was doing, I don't even need such a comment. It's self-evident from the name itself what this variable is doing.

## 1.7 Part 6: Whitespace in Python

Whitespace (no, not that Whitespace) is important in the Python language.

Some languages like C++ and Java use semi-colons to delineate the end of a single statement. Python, however, does not, but still needs some way to identify when we've reached the end of a statement.

In Python, it's the **return key** that denotes the end of a statement.

Returns, tabs, and spaces are all collectively known as "whitespace", and each can drastically change how your Python program runs. Especially when we get into loops, conditionals, and functions, this will become critical and may be the source of many insidious bugs.

For example, the following code won't run:

Python sees the indentation–it's important to Python in terms of delineating blocks of code–but in this case the indentation doesn't make any sense. It doesn't highlight a new function, or a conditional, or a loop. It's just "there", making it unexpected and hence causing the error.

This can be particularly pernicious when writing longer Python programs, full of functions and loops and conditionals, where the indentation of your code is constantly changing. For this reason, I am giving you the following mandate:

**DO NOT MIX TABS AND SPACES!!!**

If you're indenting your code using 2 spaces, *ALWAYS USE SPACES.*
If you're indenting your code using 4 spaces, *ALWAYS USE SPACES.*
If you're indenting your code with a single tab, *ALWAYS USE TABS.*

Mixing the two in the same file will cause **ALL THE HEADACHES**. Your code will crash but will be coy as to the reason why.

## 1.8 Administrivia

- **How is Assignment 1 going?**. Ask on `#questions` if you're having trouble with anything!
- **Not on Slack? Can't access JupyterHub?** Email me and we'll get it sorted out.
- **Next Tuesday we'll have a guest lecture from someone at GACRC.** Some of what we covered in the command line will come up as you learn how to run computational experiments on the GACRC hardware.

## 1.9 Additional Resources


1. Guido's PyCon 2016 talk on the future of Python: https://www.youtube.com/watch?v=YgtL4S7Hrwo
2. VanderPlas, Jake. *Python Data Science Handbook*. https://github.com/jakevdp/PythonDataScienceHandboo
3. Matthes, Eric. *Python Crash Course*. 2016. ISBN-13: 978-1593276034
4. Grus, Joel. *Data Science from Scratch*. 2015. ISBN-13: 978-1491901427