Python Programming

Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Programming is generally built on 3 steps

1. *Sequence (Do x then y)*
2. *Selection (if x then y)*
3. *Repitition (do y x times)*

Python has tools for all these as we will see.

Build-In Python Objects

Following are some python build in data types/ object types that we will learn in this course. We will see them one by one.

Object Type	**Example**
Numbers	1234, 12.34, 3+4j, Decimal(), Fraction()
Strings	‘Bobby”s’, ‘Bobby’
Lists	[1,2,[3,4]] ,
Dictionaries	{‘food’ : ‘parantha’ , ‘taste’ : ‘yum’}
Tuples	(1,’python’,’2,’,’ml’)
Sets	{‘a’, ‘b’,’c’}
Files	open(‘classroom.txt’)
Others	Booleans,None

Numbers

Integers are written as strings of decimal digits. Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an e or E and followed by an optional sign. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression.

Floating-point numbers are implemented as C “doubles” in standard CPython, and therefore get as much precision as the C compiler used to build the Python interpreter gives to doubles.

Arithmetic Operators

Let’s get started in Python by working with numbers. The basic arithmetic operations are available, and Python will follow all the usual rules of mathematics when it comes to how they work.

I can try out a calculation in a line of Python like this.

```
print(3 + 1)
```

Output:

```
4
```

This prints the result, 4. We could have done this calculation without printing it, but then we would see the result only in python shell but not in python script (we will discuss how to write script later)wouldn't see the result!

```
3 + 4
```

Output:

```
```

The symbols for addition and subtraction in Python are the usual ones, `+` and `-`, multiplication is asterisk `*` (watch out, it's not `x`) and division is forward slash `/`. Brackets for mathematics are the curved parentheses `(` and `)`. You'll also see parentheses around the content of the call to `print` that we used to show the answer to our calculation.

```
print(1 + 2 + 3 * 3)
print((1 + 2 + 3) * 3)
```

Output:

```
12
18
```

This is proof that parentheses matter!

We can go beyond the basics, too. You can raise one number to the power of another with two asterisks `**`.

```
print(3**2)
```

Output:

```
9
```

Note: There is another operator that is sometimes mistaken for the exponentiation operator, the caret: `^`. This is not for exponentiation as some programmers expect. Instead it performs a more obscure operation called bitwise xor. If you're accustomed to using the caret for exponents you might accidentally write incorrect code that produces confusing results!

Another useful operation is given by the `%` - it's the modulo operation. It gives the remainder after you've divided the first number by the second.

```
print(9 % 2)
```

Output:

```
1
```

You might also find use for integer division, denoted by `//`. It divides one integer by another, but rather than giving the exact answer it rounds the answer down to an integer. (Note: it rounds down even if the answer is negative.)

```
print(15 // 4)
print(16 // 4)
print(-5//4)
```

Output:

```
3
4
-2
```

Note: there are some differences in how division (`/`) works between Python 2 and Python 3. We will not go into them for now, and since this class is built for Python 3, we will focus on how division works in Python 3.

Try the following commands in the **Python Shell** one by one

```
print(3 + 1)

print(1 + 2 + 3 * 3)
print((1 + 2 + 3) * 3)
print(3**2)
print(9 % 2)
```

Exercise : How much is your 100 Rs worth after 7 years? (HInt : Use Compund Interest formula)

Integers and Floats

So far all of the numerical examples we've seen have been whole numbers: integers. But other numbers do exist in Python, and we need to be able to calculate with them and construct them.

```
print(3/4)
```

Output:

```
0.75
```

Here dividing one integer by another gives us a number that isn't an integer, 0.75. In Python (and computing in general) we represent such a number as a float, which is short for floating-point number.

Even if one integer divides another exactly, the result will be a float.

```
print(16/4)
```

Output:

```
4.0
```

[NOTE: This is Python 3 code. The previous versions of Python behave differently - the division of one int by another in Python 2 will yield an `int`, even if the expected result is not a whole number! This Python 2 behaviour is like integer division (`//`) in Python 3. We use Python 3 in this course.]

An operation involving an int and a float produces a float.

```
print(3 + 2.5)
```

Output:

```
5.5
```

To make an int, just give a whole number without a decimal point. Here is an int:

```
387
```

To make a float, include a decimal point! If the number itself is actually a whole number, that's OK: you don't even have to put anything AFTER the decimal point. Here are a couple of floats:

```
213.13
341.
```

Sometimes you might need to manually convert one numeric type to another, and you can do that by constructing new objects of those types with `int()` and `float()`.

```
print(int(49.7))
print(int(16/4))
print(float(3520+3239))
```

Output:

```
49
4
6759.0
```

When we convert a float to an int, the part of the number after the decimal point is cut off.

So we've seen Python's two main numeric types - integer (int) and floating-point number (float). What are they good for?

- int - there are many times when you might need to count items, or need to rely on the result of a computation being an integer. The int type is great for this.
- float - if what you're working on isn't necessarily a whole number, a float is the type you're looking for!

Floating-point numbers are approximations of the numbers they are supposed to represent. This is necessary because floats can represent an enormous range of numbers, so in order to fit numbers in computer memory, Python must use approximations. This tradeoff can sometimes have surprising results:

```
print(0.1)
print(0.1 + 0.1 + 0.1)
```

Output:

```
0.1
0.30000000000000004
```

Because the float (i.e. the approximation) for `0.1` is actually slightly more than 0.1, when we add several of them together we can see the difference between the mathematically correct answer and the one that Python creates. In most contexts these small differences are irrelevant, but it's important to know that they're there!

The Python documentation explains more about this**:**
https://docs.python.org/3/tutorial/floatingpoint.html

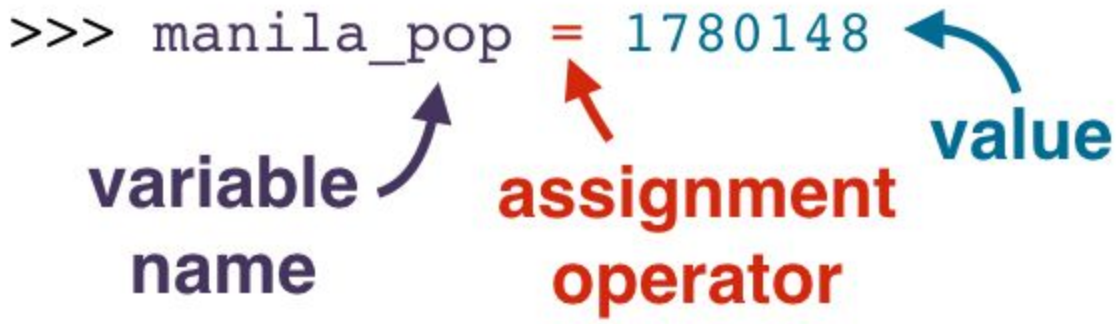
Variable

Doing arithmetic was OK, but using variables turns Python into more than just a calculator.

In this portion of the lesson we are going to learn about variables. We will look at the population of Manila (the capital of the Philippines), and we are going to do some calculations with it. We are going to accomplish that using variables. Using variables (as opposed to doing calculations on raw numbers) has many advantages, including the ability to account for changes more efficiently, as we will see later on. Let's get started!

Creating a new variable in Python is simple; here's one which stores the population of Manila.

```
manila_pop = 1780148
```



The variable name in this example is `manila_pop`. The equals sign, `=`, is the assignment operator. The value of the variable `manila_pop`is 1780148.

Assigning And Printing Variables

The order of this assignment expression is VERY important! It always goes in that same order, variable name = value. The variable_name on the left is now a name for the value given by the expression on the right. The assignment operator `=` assigns the value on

the right to the variable name on the left. (Note how this is different from writing expressions in mathematics, where `x=y` is equivalent to `y=x`)

Notice that there's no keyword for variable assignment in Python as there is in some languages, and there's no need to specify the type of the value - just go ahead and use an equals sign to assign a variable.

If you want to access that value, you can just use the name of the variable. You could, for example, print it out to the screen:

```
print(manila_pop)
```

Output:

```
1780148
```

We have used the `print` function quite a lot in this class. `print` also comes in very handy when you need to know what the value of a certain variable is- you can simply `print` that variable! Without `print`, what happens in Python, stays in Python. Most of the time, the majority of things in your program are things that happen in Python and data will just get used and passed around until it's time for the user to see something. `print` will help you to see what's going on in there, which makes it very useful for debugging when something's going wrong.

`print` is a built-in function in Python, and you'll come across more of those later on. Function calls in Python always have a pair of parentheses attached, and if there are any arguments, they go inside the parentheses. So the `print` function syntax requires a set of parentheses and the argument is put inside the parentheses. As you've already seen, if you put a variable here, the thing that gets printed is the value of that variable, not the variable name.

Note: in Python 2, users didn't need to use parentheses for printing, but in Python 3 they are required, so don't miss them!

```
print manila_pop
```

Output:

```
 File "<stdin>", line 1
    print manila_pop
                   ^
SyntaxError: Missing parentheses in call to 'print'
```

There's a SyntaxError, and a nice clear message about what went wrong. Mistakes like this are very common, but using the error message will help you to get back on track.

What's In A Name(error)?

We've been looking at the population of Manila, but imagine that we really want to find out the population density. Let's create a variable that is the result of a calculation.

```
manila_pop = 1780148
manila_pop_density = manila_pop / manila_area
```

Output:

```
NameError: name 'manila_area' is not defi
```

Uh oh - looks like another error. What's happened here? Well, I tried to divide by the variable `manila_area`, but I hadn't already assigned this as a variable name to a value,

so a NameError was raised. The error clearly gives the name that was the problem. If you want to use a variable name, you must assign a value to it first!

I'll try again:

```
manila_pop = 1780148
manila_area = 16.56
manila_pop_density = manila_pop/manila_area
print(manila_pop_density)
```

Output:

```
107496.85990338166
```

No NameError this time, and we've successfully calculated the population density of Manila. The area of Manila is 16.56 square miles so this population density is in people per square mile.

When you're naming variables there are a few things you have to watch out for:

- There are some reserved words that you cannot use for names - things like `False` and `class` which have important purposes in Python. You can find a list here: https://docs.python.org/3/reference/lexical_analysis.html#keywords . Trying to assign a value to one of these will give you a `SyntaxError`.
- Use only ordinary letters, numbers and underscores in your variable names. Start variable names with a letter or an underscore.
- It would be a bad idea to use any of the built-in identifiers for names, though this won't immediately cause an error. For example, assigning a value to `int` will not cause errors when you make the assignment, but will be really problematic you want to convert something to an int later on.

```
int = 7
int(3.0)
```

Output:

```
TypeError: 'int' object is not callable
```

- It's best to use variable names that are English words and describe what they are for as far as possible. Use underscores to separate words if you want a multiple-word variable. For example, `coconut_counter = 2`.

Assigning A Variable Again!

We already set `manila_pop` variable as the population of Manila, 1780148. What if the population of Manila changes, can we update the population data? The population is now up to 1781573. We can update Python by assigning a new value to that same variable name, which will change the value.

```
manila_pop = 1781573
print(manila_pop)
```

Output:

```
1781573
```

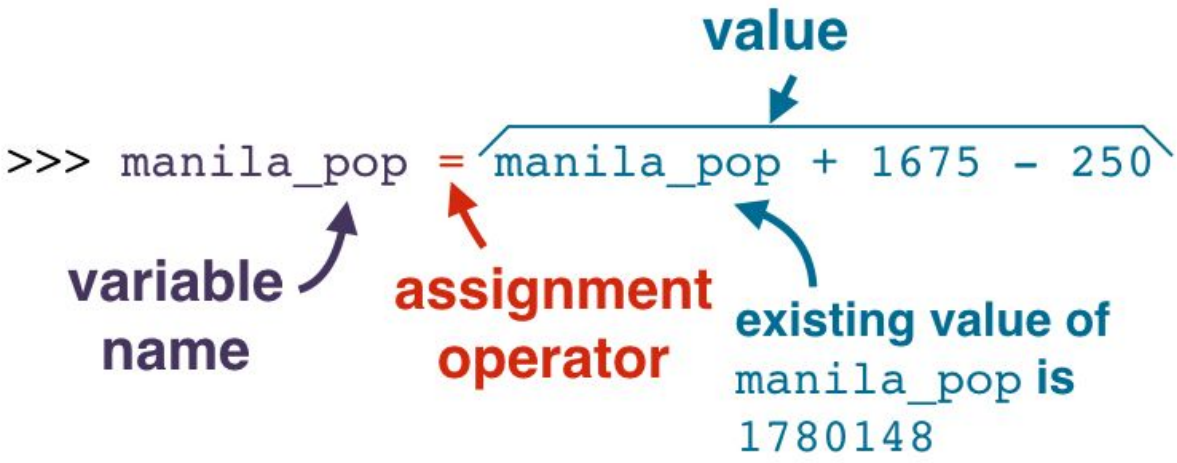
The old data has been forgotten, replaced by the new value of `manila_pop`.

We can also do this in another way, using Python to update the value. Perhaps we find out that 1675 people moved to the city and 250 moved away. We can do the calculation in Python to find the new value and assign it to the variable in one step.

```
manila_pop = manila_pop + 1675 - 250
print(manila_pop)
```

Output:

```
1780148
```



The variable `manila_pop` is to the left of the equals sign, it is assigned to the value of the whole expression on the right hand side. On the right-hand-side of the equals sign is the expression `manila_pop + 1675 -250`. The existing value of `manila_pop` is known to be 1780148.

This assignment, `manila_pop = manila_pop + 1675 - 250`, looks totally wrong if we were doing mathematics, because the variable name `manila_pop` is on both sides of the equals sign! But it works in Python code because equals sign `=` is for assignment in Python.

Reassignment Operators

Because this kind of increment and re-assign operation is very common, Python includes special operators for it:

```
manila_pop += 1675 # increase the value of manila_pop by 1675
manila_pop -= 250 # decrease the value of manila_pop by 250
manila_pop *= 0.9 # decrease the value of manila_pop by 10%
manila_pop /=  2 # approximate the female population of Manila
```

`manila_pop += 1675` is an abbreviated way of writing, `manila_pop = manila_pop + 1675`. The other reassignment operators follow the same pattern.

Exercise :

Now it's your turn to work with variables. The comments in this quiz (the lines that begin with `#`) have instructions for creating and modifying variables. After each comment write a line of code that implements the instruction.

Note that this code uses scientific notation to define large numbers. `4.445e8` is equal to `4.445 * 10 ** 8` which is equal to `444500000.0`.

```
# The current volume of a water reservoir (in cubic metres)
reservoir_volume = 4.445e8
# The amount of rainfall from a storm (in cubic metres)
rainfall = 5e6

# decrease the rainfall variable by 10% to account for runoff

# add the rainfall variable to the reservoir_volume variable

# increase reservoir_volume by 5% to account for stormwater that flows
# into the reservoir in the days following the storm

# decrease reservoir_volume by 5% to account for evaporation

# subtract 2.5e5 cubic metres from reservoir_volume to account for water
# that's piped to arid regions.

# print the new value of the reservoir_volume variable
```

Multiple assignment

It is also possible to assign two variables on a single line:

```
#These two assignments can be abbreviated
savings = 514.86
salary = 320.51

#Using multiple assignment
savings, salary = 514.86, 320.51
```

This first variable is assigned the first value after the `=`, and the second variable receives the second value. You can use this when you're assigning two closely related variables, like the width and height of an object, or its x and y coordinates.

Changing Variables

How does changing a variable affect another variable that was defined in terms of it? Let's look at an example.

Here’s the initial data about Manila’s population and population density.

```
manila_pop = 1780148
manila_area = 16.56
manila_pop_density = manila_pop/manila_area
```

```
print(int(manila_pop_density))
```

Output:

```
107496
```

Now we redefine the `manila_pop` variable:

```
manila_pop = 1781573
```

The correct answer is that the value of `int(manila_pop_density)` has not changed. This is because when a variable is assigned it is assigned to the *value of the expression* on the right-hand-side, not to the expression itself. In the line

```
manila_pop_density = manila_pop/manila_area
```

Python actually did the calculation to evaluate the expression on the right-hand-side, `manila_pop/manila_area` and then assigned the variable `manila_pop_density` to be the value of that expression. It promptly forgot the formula, only saving the result in the variable. In order to update the value of `manila_pop_density` to take into account the change in `manila_pop`, we need to run this line again:

```
manila_pop_density = manila_pop/manila_area
print(int(manila_pop_density))
```

Output:

```
107582
```

That's the new population density, after people have moved in and out of the city - all of our variables have been updated to take this into account.

Strings and Text

Programming involves more than just numbers and arithmetic, and sometimes you'll need to deal with text. To work with text in Python you will need to use a `string` - which is just a series of characters. A string is the type that holds text.

Note: If you have written code before, most likely the concept of strings is not new to you. Unless you are specifically familiar with strings in Python you will likely still benefit from this portion of the lesson, as strings in different programming languages behave differently.

You can create a string by using quotes - single or double quotes work equally well.

```
print("hello") #used double-quotes here
print('hello') #used single-quotes on this one
```

Output:

```
hello
hello
```

In each case, I printed the string `"hello"` and got the output `hello`.

I can set a variable to be a string in the same way as a number. Strings can include any characters: even spaces, punctuation and numbers.

```
welcome_message = "Hello, welcome to Udacity!"
print(welcome_message)
```

Output:

```
Hello, welcome to Udacity!
```

Strings are not numbers, but there are some operations that worked for integers and floats that will also work for strings. For example, we can use the `+` to put strings together - we call this to *concatenate* strings.

```
instructor_1 = "Philip"
instructor_2 = "Charlie"
print(instructor_1 + instructor_2)
```

Output:

```
PhilipCharlie
```

There are variables with the names of both of the instructors. We used the `+` to concatenate the two strings together to print. This is fundamentally different from numerical addition! However, notice that the two names have been squished together - we're missing a space. Python is completely literal when working with strings - we need to explicitly include spaces and punctuation if we want what we write to make sense.

```
instructor_1 = "Philip"
instructor_2 = "Charlie"
print(instructor_1 + " and " + instructor_2)
```

Output:

```
Philip and Charlie
```

This time we got a string that makes sense, putting the two instructors together. Notice that the joining string used has a space before and after the `and` in order to give proper spacing.

So, using `+` to operate on strings seems to work. Let's try another mathematical operation:

```
print(instructor_1 / instructor_2)
```

Output:

```
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

OK - here's a new kind of error! The operator `/` doesn't work for the string (`str`) type. That's probably for the best - I would hate to see the mess from dividing one instructor by another!

Quotes within Quotes

Using quotation marks to define strings presents a problem, how do we define a string that has quotation marks in it? This code does not work because the string itself includes a quotation:

```
pet_halibut = "Why should I be tarred with the epithet "loony" merely
because I have a pet halibut?"
```

Python offers two solutions to this problem. The first is to place the string in single quotes (`'`) rather than double quotes (`"`) like this:

```
pet_halibut = 'Why should I be tarred with the epithet "loony" merely
because I have a pet halibut?'
```

You can use either type of quote to define strings. Sometimes though you'll need to define a string that includes both single and double quotes. What then? In that case you can use a backslash, `\`, to escape quotes.

```
salesman = '"I think you\'re an encyclopaedia salesman"'
```

Here the string is delimited by single quotes. The single quote within the string is preceded by a backslash so that Python knows that it should be interpreted as part of the string rather than the quote that ends the string.

Quiz: Fix the Quote

The line of code in the following quiz will cause a SyntaxError, thanks to the misuse of quotation marks. First run it with Test Run to view the error message. Then resolve the problem so that the quote (from Henry Ford) is correctly assigned to the variable `ford_quote`.

We've already seen that the type of objects will affect how operators work on them. What will be the output of this code?

```
coconut_count = "34"
mango_count = "15"
tropical_fruit_count = coconut_count + mango_count
print(tropical_fruit_count)
```

Quiz: Write a Server Log Message

In this programming quiz, you're going to use what you've learned about strings to write a logging message for a server.

You'll be provided with example data for a user, the time of their visit and the site they accessed. You should use the variables provided and the techniques you've learned to print a log message like this one (with the username, url, and timestamp replaced with values from the appropriate variables):

```
Yogesh accessed the site http://petshop.com/pets/reptiles/pythons at 16:20.
```

```
username = "Kinari"
timestamp = "04:50"
url = "http://petshop.com/pets/mammals/cats"

# TODO: print a log message incorporating the strings above.
# The message should be use the same format as this one:
# "Yogesh accessed the site http://petshop.com/pets/reptiles/pythons at
16:20."
```

We've already been using one of Python's built-in functions, `print`. Another useful built-in function is `len`. The `len` function computes the length in characters of strings passed into it like this:

```
udacity_length = len("Udacity")
print(udacity_length)
```

Output:

```
7
```

The `len` function is similar to `print` in that we call it by passing a variable as the argument inside of parentheses. `len` differs from print in that it produces a value that can be stored in a variable. In the previous example the `len` function outputs the number 7 which is then stored in the `udacity_length`variable.

Quiz: `len`

Use string concatenation and the `len` function to find the length of Charlie's full name. (Yes, she is unhappy with her parents' choices…!) Store that length in the `name_length` variable. Don't forget that there are spaces in between the different parts of a name!

```
given_name = "Charlotte"
middle_names = "Hippopotamus"
family_name = "Turner"

name_length = #todo: calculate how long this name is

driving_licence_character_limit = 28
print(name_length <= driving_licence_character_limit)
```

We've just used the `len` function to find the length of strings. What does the len function return when we give it the integer `835` instead of a string?

What Type Is This Object?

We've talked a bit about types so far, and we've discussed four of them in total:

- **`int` (integer, for whole numbers)**
- **`float` (for numbers that aren't necessarily whole numbers)**
- **`bool` (boolean, for True and False values)**
- **`str` (string, for text)**

In Python, every object you encounter will have a type. An object's type defines which operators and functions will work on that object and how they work!

You can check the type of an object directly using the built-in function **`type()`.**

```
print(type(633))
print(type("633"))
print(type(633.0))
```

Output:

```
int
str
float
```

This is essentially the same number coded in three different ways, and so they have three different types, each with their own set of behaviours.

As a side-note, here we're calling a function print() on the output of another function, type(). We use parentheses to define the order in which functions get run: what's contained in one set of parentheses needs to be evaluated first before being given as input to the next function. Here the type() function is run first and then its output is print()-ed.

What type does this object have? `"12"`

What type does this object have? `12.3`

What type does this object have? `len("my_string")`
What type does this object have? `"hippo" *12`

Choosing Types

Different types have different properties, and when you're designing a computer program, you'll need to choose the types for your data based on how you're going to use them. For example, if you want to use a number as a part of a sentence, it'll be easiest if that number is a string. If you want to encode a true/false value, it will be much easier to manipulate as a boolean than a string!

You might also have situations where you don't control the type of data that you receive but still need to use it. The good news is you can create new objects from old, and change the type in the process. Look at some examples:

Creating an int from a float and assigning it to a variable `count`, printing `count` and its type.

```
count = int(4.0)
print(count)
print(type(count))
```

Output:

```
4
<class 'int'>
```

Making a string from a number:

```
house_number = 13
street_name = "The Crescent"
town_name = "Belmont"

print(type(house_number))
address = str(house_number) + " " + street_name + ", " + town_name
print(address)
```

Output:

```
<class ‘int’>
13 The Crescent, Belmont
```

You can also build a number from a string:

```
grams_of_sugar = float("35.0")
print(grams_of_sugar)
print(type(grams_of_sugar))
```

Output:

```
35.0
<class ‘float’>
```

Quiz: Total Sales

In this quiz, you'll need to change the types of the input and output data in order to get the result you want.

Mon_sales = "235"
Tue_sales = "256"
Wed_sales = "564"
Thu_sales = "890"
Fri_sales = "589"

Calculate and print the total sales for the week from the data provided. Print out a string of the form `"This week's total sales: xxx"`, where xxx will be the actual total of all the numbers. You'll need to change the type of the input data in order to calculate that total.