Python Home Page (https://www.kaggle.com/learn/python)

Intro

This course covers the key Python skills you'll need so you can start using Python for data science. The course is ideal for someone with some previous coding experience who wants to add Python to their repertoire or level up their basic Python skills. (If you're a first-time coder, you may want to check out these "Python for Non-Programmers" learning resources (https://wiki.python.org/moin/BeginnersGuide/NonProgrammers).)

We'll start with a brief overview of Python syntax, variable assignment, and arithmetic operators. If you have previous Python experience, you can skip straight to the hands-on exercise (https://www.kaggle.com/kernels/fork/1275163).

Hello, Python!

Python was named for the British comedy troupe Monty Python (https://en.wikipedia.org/wiki/Monty_Python), so we'll make our first Python program an homage to their skit about Spam?

Just for fun, try reading over the code below and predicting what it's going to do when run. (If you have no idea, that's fine!)

Then click the "output" button to see the results of our program.

In [1]:

Output

```
spam_amount = 0
print(spam_amount)

# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more servings
of Spam)
spam_amount = spam_amount + 4

if spam_amount > 0:
    print("But I don't want ANY spam!")

viking_song = "Spam " * spam_amount
print(viking_song)
```

There's a lot to unpack here! This silly program demonstrates many important aspects of what Python code looks like and how it works. Let's review the code from top to bottom.

In [2]:

```
spam_amount = 0
```

Variable assignment: Here we create a variable called `spam_amount` and assign it the value of 0 using `=`, which is called the assignment operator.

> **Aside**: If you've programmed in certain other languages (like Java or C++), you might be noticing some things Python *doesn't* require us to do here:
>
> - we don't need to "declare" `spam_amount` before assigning to it
> - we don't need to tell Python what type of value `spam_amount` is going to refer to. In fact, we can even go on to reassign `spam_amount` to refer to a different sort of thing like a string or a boolean.

In [3]:
```
print(spam_amount)
```

```
0
```

Function calls:. `print` is a Python function that displays the value passed to it on the screen. We call functions by putting parentheses after their name, and putting the inputs (or *arguments*) to the function in those parentheses.

In [4]:
```
# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more servings of Spam)
spam_amount = spam_amount + 4
```

The first line above is a **comment**. In Python, comments begin with the `#` symbol.

Next we see an example of reassignment. Reassigning the value of an existing variable looks just the same as creating a variable - it still uses the `=` assignment operator.

In this case, the value we're assigning to `spam_amount` involves some simple arithmetic on its previous value. When it encounters this line, Python evaluates the expression on the right-hand-side of the `=` (0 + 4 = 4), and then assigns that value to the variable on the left-hand-side.

In [5]:
```
if spam_amount > 0:
    print("But I don't want ANY spam!")

viking_song = "Spam Spam Spam"
print(viking_song)
```

```
But I don't want ANY spam!
Spam Spam Spam
```

We won't talk much about "conditionals" until later, but, even if you've never coded before, you can probably guess what this does. Python is prized for its readability and the simplicity.

Note how we indicated which code belongs to the `if`. `"But I don't want ANY spam!"` is only supposed to be printed if `spam_amount` is positive. But the later code (like `print(viking_song)`) should be executed no matter what. How do we (and Python) know that?

The colon (`:`) at the end of the `if` line indicates that a new "code block" is starting. Subsequent lines which are **indented** are part of that code block. Some other languages use `{`curly braces`}` to mark the beginning and end of code blocks. Python's use of meaningful whitespace can be surprising to programmers who are accustomed to other languages, but in practice it can lead to more consistent and readable code than languages that do not enforce indentation of code blocks.

The later lines dealing with `viking_song` are not indented with an extra 4 spaces, so they're not a part of the `if`'s code block. We'll see more examples of indented code blocks later when we define functions and using loops.

This code snippet is also our first sighting of a **string** in Python:

```
"But I don't want ANY spam!"
```

Strings can be marked either by double or single quotation marks. (But because this particular string *contains* a single-quote character, we might confuse Python by trying to surround it with single-quotes, unless we're careful.)

In [6]:

```
viking_song = "Spam " * spam_amount
print(viking_song)
```

```
Spam Spam Spam Spam
```

The `*` operator can be used to multiply two numbers (`3 * 3` evaluates to 9), but amusingly enough, we can also multiply a string by a number, to get a version that's been repeated that many times. Python offers a number of cheeky little time-saving tricks like this where operators like `*` and `+` have a different meaning depending on what kind of thing they're applied to. (The technical term for this is operator overloading (https://en.wikipedia.org/wiki/Operator_overloading))

Numbers and arithmetic in Python

We've already seen an example of a variable containing a number above:

```
In [7]:
spam_amount = 0
```

"Number" is a fine informal name for the kind of thing, but if we wanted to be more technical, we could ask Python how it would describe the type of thing that `spam_amount` is:

```
In [8]:
type(spam_amount)
```

```
Out[8]:
int
```

It's an `int` - short for integer. There's another sort of number we commonly encounter in Python:

```
In [9]:
type(19.95)
```

```
Out[9]:
float
```

A `float` is a number with a decimal place - very useful for representing things like weights or proportions.

`type()` is the second built-in function we've seen (after `print()`), and it's another good one to remember. It's very useful to be able to ask Python "what kind of thing is this?".

A natural thing to want to do with numbers is perform arithmetic. We've seen the `+` operator for addition, and the `*` operator for multiplication (of a sort). Python also has us covered for the rest of the basic buttons on your calculator:

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`

One interesting observation here is that, whereas your calculator probably just has one button for division, Python can do two kinds. "True division" is basically what your calculator does:

In [10]:
```
print(5 / 2)
print(6 / 2)
```

```
2.5
3.0
```

It always gives us a `float`.

The `//` operator gives us a result that's rounded down to the next integer.

```
In [11]: print(5 // 2)
         print(6 // 2)
```

```
2
3
```

Can you think of where this would be useful? You'll see an example soon in the coding challenges.

Order of operations

The arithmetic we learned in primary school has conventions about the order in which operations are evaluated. Some remember these by a mnemonic such as **PEMDAS** - **P**arentheses, **E**xponents, **M**ultiplication/**D**ivision, **A**ddition/**S**ubtraction.

Python follows similar rules about which calculations to perform first. They're mostly pretty intuitive.

```
In [12]: 8 - 3 + 2
```

```
Out[12]: 7
```

```
In [13]: -3 + 4 * 2
```

```
Out[13]: 5
```

Sometimes the default order of operations isn't what we want:

In [14]:
```python
hat_height_cm = 25
my_height_cm = 190
# How tall am I, in meters, when wearing my hat?
total_height_meters = hat_height_cm + my_height_cm / 100
print("Height in meters =", total_height_meters, "?")
```

```
Height in meters = 26.9 ?
```

Parentheses are your useful here. You can add them to force Python to evaluate sub-expressions in whatever order you want.

In [15]:
```python
total_height_meters = (hat_height_cm + my_height_cm) / 100
print("Height in meters =", total_height_meters)
```

```
Height in meters = 2.15
```

Builtin functions for working with numbers

`min` and `max` return the minimum and maximum of their arguments, respectively...

In [16]:
```python
print(min(1, 2, 3))
print(max(1, 2, 3))
```

```
1
3
```

`abs` returns the absolute value of it argument:

In [17]:
```
print(abs(32))
print(abs(-32))
```

```
32
32
```

In addition to being the names of Python's two main numerical types, `int` and `float` can also be called as functions which convert their arguments to the corresponding type:

In [18]:
```
print(float(10))
print(int(3.33))
# They can even be called on strings!
print(int('807') + 1)
```

```
10.0
3
808
```

Your Turn

Now is your chance. Try your **first Python programming exercise (https://www.kaggle.com/kernels/fork/1275163)**

Python Home Page (https://www.kaggle.com/learn/python)

Have questions or comments? Visit the Learn Discussion forum (https://www.kaggle.com/learn-forum/161283) to chat with other Learners.