Introduction to Python

Command line, printing, comments, math operators, error types, intro to strings

Hello world

Use the command line for these.

```
>>> print("Hello world!")
Hello world!
>>> print('Hello world!')
Hello world!
```

- Hello world is called a string and needs quotes - single or double.
- print() is a pre-defined Python function that displays arguments on the screen.
- Whitespace doesn't matter except indentations. Single spaces between operators is convention.

```
>>> 2 + 2
4
```

Comments

Comments let you leave notes, too!

```
>>> 2 + 2    # This is an expression. 2 is a value, + is an operator.
4           # The expression evaluates to 4
```

Mathematical operators

Python operators (in order of precedence):

Operator	Operation	Example	Evaluates to:
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division	22 // 8	2
/	Division	22 /8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	5 + 2	7

Be careful of order of operations. Parentheses are your friends!

```
>>> 2 + 3 * 6 # What do you think this evaluates to?
20
```

It's safer to use parentheses to be sure you get what you what - and make it easier to read.

```
>>> 2 + (3 * 6)
20

>>> (2 + 3) * 6
30
```

Play with the Console for a bit. Notice any errors you get - or other unexpected behavior.

Data types

Data can have four main data types:

- int - integers
- float - floating point numbers (decimals)
- str - strings (text), surround with quotes
- bool - boolean (True, False)

To check the type, use function type()

```
>>> type(3)
<class 'int'>
```

To convert between types, use type functions:

- int()
- float()
- str()
- bool()

(Note that this only works when the conversion makes sense.)

```
int('Hello')    # This will give an error!

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
```

Types of errors

You will see a lot of errors, especially at the beginning. If Python doesn't understand what you want, it doesn't guess. It just spits out an error message.

Fortunately, these messages are often (but not always) helpful in figuring out what went wrong.

Syntax errors

Syntax errors are errors in the language of Python. Just like there are rules for punctuation and capitalization in English, Python has its own rules. In the example below, the string is missing one the quotes. Python won't guess what is meant. It just quits and spits out a nasty message (aka 'nastygram').

```
>>> print('hello)

  File "<stdin>", line 1
    print('hello)
                ^
SyntaxError: EOL while scanning string literal
```

Runtime errors

Runtime errors appear when you run the program (as opposed to errors that keep the program from running). They are also called *exceptions* because they show something exceptional (ie: bad) has happened. Python understands what you are telling it to do, but there is a problem doing it. An example is dividing by 0.

```
>>> print(5/0)

Traceback (most recent call last):
  File "/tmp/sessions/bca0f883a31d95c8/main.py", line 3, in <module>
    print(5/0))
ZeroDivisionError: division by zero
```

Semantic errors

These can be the hardest to find. The program runs correctly as written, but you made a mistake writing it so it doesn't do what you expect. There will not be any error messages for this - you will need to test multiple scenarios to be sure it is functioning correctly.

An example is using + instead of * in a calculation or putting the wrong variable in somewhere.

Variables

You can assign values to variables. Unlike in math, you can reassign those values throughout the program.

```
>>> x = 3
>>> print(x)
3
>>> x = 'cat'    # x has been reassigned to 'cat'
>>> print(x)
'cat'
```

User input

You can let the user interact with the program by using the input() function. Note that input values are strings by default, not matter what the input. You can change that using functions that change the data type: int(), float(), str().

```
>>> x = input()   # User enters the number 5
>>> type(x)
<class 'str'>    # Even though the user enters a number, python treats it as a string.

>>> y = int(x)    # int(x) converts the string '5' to the integer 5
>>> type(y)
<class 'int'>    # Checking the type of y confirms it is an integer.
```

Note that this works too:

```
>>> x = input()   # User enters the number 5
>>> type(x)
<class 'str'>    # Even though the user enters a number, python treats it as a string.

>>> x = int(x)    # int(x) converts the string '5' to the integer 5 and reassigns it back to x.
>>> type(x)
<class 'int'>    # Checking the type of y confirms it is an integer.
```

So does this:

```
x = int(input())    # Be careful - if the user enters a word, an error will occur.
```

Console vs Editor

So far, you have been typing commands into the console. The console executes each line when you press 'enter'. In the editor, you can write multiple lines of code and execute them with a single 'run' command. An important difference to note at this point is that function results display automatically in the console but you need to `print` them in the editor.

Console:

```
>>> type('cat')
<class 'str'>
```

Editor:

```
print(type('cat'))
```

- After running:

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

Working with strings

Strings are everywhere in programming! Almost everything that isn't a number (int or float) is a string.

Three examples (among many) of string usage are:

- collecting user input
- displaying words on the screen (instructions, for example)
- working with data that includes words (addresses, names, etc)

Reminder: strings are surrounded by '' or "".

Examples:

```
>>> name = input("Enter your name: ")    # collects user name (as a string)
 and assigns it to the variable called name
```

```
 print('Hello')    # Hello is a string
```

Sticking strings together (concatenation)

Adding numbers together computes the sum of those numbers. Adding strings together sticks them together.

```
>>> print()'Hello' + 'world')
Helloworld
```

Note the strings are stuck to each other with no space. If you want a space you can include it as part of a string or you can use separate the strings with a comma. (There are other options, too.) How you do it will depend on what you are doing.

```
>>> print()'Hello ' + 'world')    # added space at the end of Hello
Hello world

>>> print()'Hello' , 'world')    #    printed each string in sequence
Hello world
```

More on strings next time!