

Introduction to Python



Intro

The Dark Side

- We've seen a lot of client-side stuff: HTML, CSS, & JS. There's even more stuff to learn!
- It's time for us to spend some time on the other, more mysterious side...the server side!
- There are tons of languages we could use to write server-side code with:
 - *Ruby*
 - *JS (Node)*
 - *PHP*
 - *Java*
- But we'll be working with Python! (and eventually Node)

The Game Plan

- We'll start by learning basic Python syntax: variables, loops, functions, etc.
- Then we'll move on to Object Oriented Programming in Python
- We'll learn how to create our own servers using Python!
- Then it's on to Python testing
- We'll take a detour to learn SQL and see how to connect to a DB using Python
- We'll cover authentication and deployment as well

Why Python?

- It's fast, powerful, and widely used
- "high level": express concepts at a high level (*a little more than JS*)
- Super clean syntax!
- Runs on servers (*but not in a browser*)
- Particularly used for data science, machine learning, making servers, etc

(This comic is from the days of Python 2; in modern Python, that would be `print("Hello, world")` , with parentheses.

But what about server-side JS?

- Yes, you could use Node JS to write a server, connect to a DB, etc.
- (and we will be doing just that later on)
- But we're starting with Python because...

Why Not Node?

- Learning a 2nd language helps you see many of the similarities between languages
- It also helps you better understand what makes each language unique
- Learning exclusively full-stack JS is a recipe for misconceptions
- We want to force you out of your comfort zone a little bit, because learning new tools is a HUGE part of being a developer

Python Versions

Python 2	Python 3
<ul style="list-style-type: none">• Latest is 2.7• What some people still use• What comes by default on OSX	<ul style="list-style-type: none">• Latest is 3.7• Slightly different language & syntax• What we'll use at Rithm

Installing Python

Head over to <https://www.python.org/downloads/>

Test that it works: in a new *Terminal* window

```
$ which python
```

Install another Python utility: `ipython`:

```
$ pip3 install ipython
```

Interactive Python

IPython is a program for interactive exploring of Python

```
$ ipython
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
Type 'copyright', 'credits' or 'license' for more information
Python 3.7.0: An enhanced Interactive Python. Type '?' for help.

In [1]: print("Hello, World!")
Hello, World
```

(Control-D to exit)

Printing

```
print(value, value, ...)
```

- Puts spaces between values
- Puts return character ("newline") at the end

```
x = "awesome"
print("Python is", x)
```

Indentation

In many programming languages, you use `{` and `}` to show blocks:

```
if (age >= 18) {
  console.log("Please go vote!");
  registerToVote();
}
```

Programmers also tend to indent this code, but that's just visually prettiness.

This would work the same:

```
if (age >= 18) {
  console.log("Please go vote!");
  registerToVote();
}
```

(That is so ugly. Please don't do that.)

In Python, you don't use `{` / `}` for blocks; the indentation is what matters:

```
if age >= 18:
    print("Please go vote!")
    register_to_vote()
```

That's very different than:

```
if age >= 18:
    print("Please go vote!")
register_to_vote()
```

In JS, people often use 2 or 4 spaces for indentation (styles vary)

In Python, everyone agrees: it should always be 4 spaces

Variables

- Python variable name style is *like this* (lower-snake-case)
- There is no keyword for declaring variables; ie no **let** or **var**
- No specific way to make un-re-bindable like **const**
 - It's good style to write constants **LIKE THIS**
- "Lexical function scoped"

```
x = 42

def my_function():
    x = 12
    print(x) # 12

print(x) # 42
```

Strings

- Like JS, can use `"` or `'` as delimiters
- Can be multi-line by using triple-quotes: `"""` or `'''`
- Can interpolate expressions with *f-strings*:

```
food = "cheese"
print(f"I love {food}")
```

Numbers

Very much like JavaScript!

- Separate types for integers (can be any size) or floating-point
 - In JS, there are only floating-point numbers
 - Separate type for complex numbers
- `.` , `.` , `.` , `.` , `.` / (true division), `//` (integer division)
- `%` (modulo: remainder after division)
- Dividing by zero is an error (JS: is *Infinity*, except 0/0, which is *NaN*)
- Can use `+` and `*` on strings: `"cat" + "food"` or `"yay" * 3`

Lists

Like JS arrays:

- ordered
- can be heterogeneous: `[1, "apple", 13.5]`

Equality

JavaScript	Python
<ul style="list-style-type: none">• <code>==</code> loose equality<ul style="list-style-type: none">• <code>7 == "7"</code>• <code>===</code> strict equality<ul style="list-style-type: none">• <code>7 === "7" // false</code>• Objects & arrays only equal when same identity	<ul style="list-style-type: none">• <code>==</code> equality (strict about types)<ul style="list-style-type: none">• <code>7 == "7" # False</code>• Structures with same items are equal<ul style="list-style-type: none">• <code>[1, 2, 3] == [1, 2, 3]</code>• Use <code>is</code> to check obj identity<ul style="list-style-type: none">• <code>[1, 2] is [1, 2] # False</code>

Truthiness

- In JS, these things are falsy:
 - `0` , `0.0` , `""` , *undefined*, *null*, *NaN*, *false*
- In JS, these things are (perhaps unexpectedly) truthy:
 - `[]` , `{}`
- In Python, these things are falsy:
 - `0` , `0.0` , `""` , *None*, *False*
 - `[]` (empty list), `{}` (empty dictionary), `set()` (empty set)
- In Python, these things are truthy:
 - Any non-empty string, non-empty list/dict/set, non-0 number
 - *True*

And/Or/Not

- JS: `&&` , `||` , `!`
- Python: `and` , `or` , `not`
- Just like in JS, these "short circuit"

If

```
if grade == "A":
    print("awesome job!")

elif grade == "F":
    print("ut oh!")

else:
    print("don't worry too much")
```

(parens around condition aren't required, unlike JS)

```
if age >= 18:
    if unregistered:
        print("please register")

    else:
        print("keep voting!")

else:
    print("Wait a bit")
```

Ternary

```
JavaScript
let msg = (age >= 18) ? "go vote!" : "go play!"

Python
msg = "go vote!" if (age >= 18) else "go play!"
```

(in both, parens are optional but often helpful)

Loops

While Loops

```
count = 10

while count > 0:
    print(count)
    count = count - 1 # or "count -= 1", but not "count--"

print("Liftoff!")
```

For Loops

Python for loops are like JS *for...of* loops:

```
for snack in ["Peanut", "Twizzler", "Mars Bar"]:
    print("I ate a", snack)
```

To loop 5 times:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

Can also use *range()* function:

```
for num in range(5): # makes [0, 1, 2, 3, 4]
    print(num)
```

Functions

```
def add_numbers(a, b):
    sum = a + b
    print("doing math!")
    return sum
```

Functions that don't explicitly *return* return *None*

Can pass arguments by name:

```
def order_pizza(size, flavor):
    print(f"size: {size} with {flavor} topping")

order_pizza("large", "mushroom")

order_pizza(size="small", flavor="sausage")

# Same thing
order_pizza(flavor="sausage", size="small")
```

Can provide defaults for parameters:

```
def send_invite(name, city="SF", state="California"):
    print(f"mailing invitation to {city}, {state}")

send_invite("Jenny", "Portland", "Oregon")

send_invite("Joel")
```

Providing too many/too few arguments is an error (in JS, this is ignored / becomes *undefined*):

```
def add_three_numbers(a, b, c):
    return a + b + c

add_three_numbers(10, 20, 30) # 60, yay!

add_three_numbers(10, 20) # error!

add_three_numbers(10, 20, 30, 40) # error!
```

Comments and Docstrings

- `#` :rest of line is comment (use to explain complex code)
- String as very first thing in file/function is "docstring"
 - Use to document what the function/file does
 - Shown when you ask for: `help(some_function)`

```
def add_limited_numbers(a, b):
    """Add two numbers, making sure sum caps at 100."""

    sum = a + b

    # If this required explanation, comment like this

    if sum > 100:
        sum = 100

    return sum
```

Modes

Running a Source File

```
$ python3 mygame.py
You win! Your score is 10

$ python3 mygame.py
```

- runs Python
- loads *mygame.py*
- executes the code
- returns to the terminal when done.

Running in IPython

```
$ ipython
In [1]: %run mygame.py
```

- runs *mygame.py*
- stays in IPython, variables are still set

Play in the Console

It's. The. Best. Way. To. Learn.

Good idea: open a console at the same time as your editor!

Getting Help

`dir()`

"Show me the methods and attributes of this object"

```
['__add__', '__radd__', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

Note: `__methods__`

You'll notice many objects provide a lot of methods that have names starting and ending with double underscores (Python programmers often call these "special methods" or "dunder [for 'double-underscore'] methods").

These aren't methods you call directly (ie, you wouldn't ever say `mylist.__add__(0)`) – instead, these work behind-the-scenes to support other operations of the object.

Generally, you can ignore them when examining an object.

help()

"Show me help about how to use this object"

```
>>> help()

q to quit that
```