**Python!**

**Tentative schedule:**

Week 1:

Fundamentals:

Spend 1-2 days learning syntax. Syntax simply means the vocabulary of the language.

Practice the fundamentals of programming that you've already been introduced to in your morning algorithms.

Intro to Flask:

Learn how to make a web server using the Flask framework. You'll create web pages that can accept data sent from a client and return a response. This quick intro will take about 2 days.

MySQL:

Spend 1-2 days to learn about MySQL, a commonly used relational database.

Learn to create visual representations of your database.

Formulate SQL queries using database creation and management software.

Week 2:

Flask + MySQL:

Extend your use of Flask to include data persistence using a database as storage.

Challenge yourself to complete The Wall, an assignment that brings together all the skills and tools you've learned so far.

Spend two days completing this section.

Object Oriented Programming:

Learn OOP, a style of organizing your code that is prevalent in web development.

Don't underestimate how important it is to develop a solid understanding of OOP, which is a pivotal skill. This is why you spend the rest of your time at the bootcamp building on your understanding.

Frameworks like Django, Rails, Angular, and iOS are built using an object oriented design pattern.

Django:

The framework you've all been waiting for!

You will learn and use code modularization to organize your code according to the purpose it serves.

Learn to build a project in Django.

Week 3:

Django:

Continue with Django for the rest of the week leading up to your exam.

Learn to store data in a SQLite database.

Learn to use a object relational mapping (ORM) to communicate with your database.

Learn to add a MySQL database to your project.

Deployment:

Learn to move your code (previously hosted on your computer) to the web so everyone can see the product of your hard work!

You will be required to deploy your belt exam.

Belt Exam:

Friday of week 3, you'll be ready to take your belt exam.

Your instructor will give you more details as the date approaches.

Try not to worry about this for now. The exam is low-pressure and can be retaken as many times as you need.

The exam is open-web – meaning you have all of your previous code and the entire internet at your disposal!

Week 4:

Retake your belt exam

You'll have this week to retake the belt exam as many times as you need

Do a project:

After passing your belt exam, you're ready to do a project!

Your project is your motivation to earn your belt. Complete your belt quickly in order to spend as much time as possible working on a fun project. Ask your for project ideas if you need help.

Create anything you want using your new skills. The sky's the limit!

Why Python?

* Readability
* Libraries
* Community
* Scope
* Ease

Core Philosophy:

* Beautiful is better than ugly
* Explicit is better than implicit
* Simple is better than complex
* Complex is better than complicated
* Readability counts

Why Python is Good for You:

* Popular
* Shallow learning curve
* Minimal setup
* Understandable
* Fast development cycle

Issues with running Python script in bash shell:

1. Python Shell hangs on initialization

Resolution: Adding an alias to your .bashrc file.

1. Print statements do not appear.

Workarounds: (1) flush the output buffer after each print statement (i.e. “sys.stdout.flush()”); (2) if running code from document using “python filename.py”, use “python –u filename.py”.

**When creating a variable in python, you don’t need to use “var” like in Javascript.**

**“Array” in Javascript == “List” in Python**

**“Console.log” in Javascript = “Print” in Python**

For comments:

* Single line - #
* Multiple lines – Triple quotations (“”” “”” or ‘’’ ‘’’)

Data Types:

Primitive data types:

* Boolean values
* Numbers
* Strings

Composite types:

* Tuples: type of data that is immutable and can hold a group of values. Tuples can contain mixed data types.
* Lists: A type of data that is mutable and can hold a group of values. Usually meant to store a collection of related data.
* Dictionaries: A group of key-value pairs. Dictionary elements are indexed by unique keys which are used to access values.

Strings:

Print inserts a space between elements separated by a comma.

Concatenate the contents into a new string with the help of +.

Lastly, you can use curly brackets - {} - and the string **.format()** method to inject variables into your string - this is known as **string interpolation**.

first\_name = "Zen"

last\_name = "Coder"

print "My name is {} {}".format(first\_name, last\_name)

As you read other people's code, you may see a different method of string interpolation. It is a lesser-used and soon-to-be deprecated method that you should know about, but will not need to use.

hw = "hello %s" % 'world'

print hw

# the output would be:

# hello world

**Built-in methods for strings:**

* **<string>.capitalize()** – create a copy of the string with only its first character capitalized
* **<string>.lower()** – return a copy of string converted to all-lowercase
* **<string>.swapcase()** – return a copy of a string with uppercase characters converted to lower case, and vice versa
* **<string>.upper()** – return a copy of string converted to all uppercase
* **<string>.find(<substring>)** – determine if specified string occurs in a given string and returns the index or location of where substring occurred. If not found, return -1.
* **<string>.replace(<old>, <new> [, max])** – returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first number of times indicated in max will you have the substring old replaced.

**Built-in methods for Lists:**

* **<list>.append(<new\_element>)** – appends new item onto the end of the given list
* **<list>.insert(<index>, <new\_element>)** – inserts a new item into the list at the given index
* **<list>.remove(<element>)** – remove the first item from the list whose value is provided. Errors if the element does not exist.
* **<list>.pop(<optional\_index>)** – remove the item at a given position; if the position is not given, it will remove the last entry from the list.
* **<list>.sort()** – sorts the elements in a list in ascending order by numerical value, or alphabetically in the case of strings.
* **len()** – returns the number of items in a sequence
* **max()** – returns the largest item in the sequence
* **min()** – returns the smallest item in the sequence. Comparing items of differing types: **all numbers < all dictionaries < all lists < all strings < all tuples**.
* **any()** – returns True if there exists any item in the sequence which is True
* **all()** – returns True if all items in the sequence are True

**Conditional statements:**

# if statement:

**if <condition>:**

# do something

# if-else statement:

**elif <condition>:**

# do something

**else:**

# do this instead

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. | (1 == 2) is not true.  (1 == 1) is true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (1 != 2) is true. |
| <> | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (1 <> 2) is true. This is similar to != operator.\* |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (1 > 2) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (1 < 2) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (1 >= 2) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (1 <= 2) is true. |
| and | Checks each expression on the left and right. If both are true then this evaluates true. If either or both expressions are false then this is false | (1 <= 2 and 2 <= 3) is true.  (1 <= 2 and 2 >= 3) is false.  (1 >= 2 and 2 >= 3) is false. |
| or | Checks each expression on the left and right. If either of the expressions are true then this evaluates true. If both expressions are false then this is false. | (1 <= 2 or 2 >= 3) is true.  (1 <= 2 or 2 >= 3) is true.  (1 >= 2 or 2 >= 3) is false. |
| not | Reverses the true-false value of the operand | not(true) is false.  not(false) is true.  not(1 >= 2) is true.  not(1 =< 2) is false.  not(1 <= 2 and 2 =< 3) is false.  not(1 >= 2 or 2 >= 3) is true. |

**\*Note:** != can also be written <>, but this is an obsolete usage kept for backwards compatibility only. New code should always use !=.

For loop:

**for** <counter> **in** <sequence or range>:

# do something

While loop:

**while** <expression>:

# do something

Loop control:

* **break** - The break statement exits the current loop prematurely, resuming execution at the first post-loop statement
* **continue** - The continue statement returns the control to the beginning of the loop. The continue statement rejects -- or skips -- all the remaining statements in the current iteration of the loop, and continues normal execution at the top of the loop.
* **pass** - The pass statement is used when a statement is required syntactically but you do not want any command or code to execute. The pass statement is a null operation; nothing happens when it executes. The pass is *almost never seen in final production*, but can be useful *in places where your code has not been completed yet*.
* **else** - There are certain conditions that we give for every loop that we have, but what if the condition was not met and we still would like to do something if that happens? We can then use else. Yes, that is right, else in a loop.

Examples of using “else” in a loop…

x = 3

y = x

while y > 0:

print y

y = y - 1

**else:**

print "Final else statement"

Output will be: 3 2 1 “Final else statement”

x = 3

y = x

while y > 0:

print y

y = y - 1

if y == 0:

*break*

**else:**

print "Final else statement"

Output will be: 3 2 1

**Functions:**

Python syntax:

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

**Functions in Python are known as "First Class"**

They can go wherever they want:

* stored as standalone functions
* defined inside Classes/Instances (known as methods)
* passed as arguments to functions (known as callbacks)
* returned from functions (creating a closure)
* anonymous which are stored in variables/used only one time (known as lambdas)

We *define* **parameters**. We *pass in* **arguments** into functions.

Functions *can return any of the data types* - strings, numbers, lists, tuples, dictionaries and even other functions!

**Tuples:**

A **Tuple** is a container for a fixed sequence of data objects. The *name comes from the Latin suffix for multiples*: double, triple, quadruple, quintuple. Tuples are sequences, just like lists. The only difference is that *tuples can't be changed* -- that is, tuples are immutable. Also, while lists are defined using square brackets, *tuples use parentheses*. Creating a tuple is as simple as declaring different comma-separated values. Optionally you can put these values between parentheses.

Tuples are useful for representing what other languages often call *records* — *some related information that belongs together*, like your student record. There is no description of what each of these fields means, but we can guess. A tuple lets us “chunk” together related information and use it as a single thing.

Once Python has created a tuple in memory, it cannot be changed. But we can add and slice tuples.

Examples:

dog = ("Canis Familiaris", "dog", "carnivore", 12)

dog = dog + ("domestic")

#result is...

#("Canis Familiaris", "Dog", "carnivore", 12, "domestic")

dog = dog[:3] + ("man's best friend") + dog[4:]

#result is...

#("Canis Familiaris", "Dog", "carnivore", "man's best friend", "domestic")

Python has a very powerful *tuple assignment* feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

In **tuple packing**, the values on the right are ‘packed’ together in a tuple and stored in a variable:

value = ("Michael", "Instructor", "Coding Dojo") #tuple packing

In **tuple unpacking**, the values in a tuple on the right are ‘unpacked’ into the variables/names on the left:

value = ("Michael", "Instructor", "Coding Dojo")

(name, position, company) = value #tuple unpacking

print name

print position

print company

**Built-in Tuple Functions:**

* **len()** – returns the number of items in a sequence
* **max()** – returns the maximum item in the sequence
* **min()** – returns the minimum item in the sequence
* **sum()** – sums the individual items
* **any()** – return *True* if there exists any item in the tuple which is TRUE.
* **all()** – return *True* if all items are TRUE
* **enumerate()** – Iterate through the tuple returning 2-tuples of (index, item). This function “enumerates” all the items in a sequence: It provides a number and each element of the original sequence in a 2-tuple.
* **sorted()** – iterate through the tuple in sorted order. Note: the returned collection is a sorted **list**, not a tuple.
* **reversed()** – iterate through the tuple in reverse order. Note: the return value is generic <reversed object> and must be fed into the tuple() or list() constructor to create one of those objects.

**Dictionaries:**

A **Dictionary** is another mutable set type that can store any number of Python objects, including other set types. Dictionaries consist of pairs (called items) of keys and their corresponding values. While this data structure is known as a dictionary in Python, you'll see the same structure referred to as an **associative array or hash table** in other languages. In general, hash table is the most generic term.

General summary of characteristics of a Python dictionary:

* A dictionary is an unordered collection of objects.
* Values are accessed using a key.
* A dictionary can shrink or grow as needed.
* The contents of dictionaries can be modified.
* Dictionaries can be nested.
* Sequence operations such as slice cannot be used with dictionaries.

Example: Creating dictionary

weekend = {"Sun": "Sunday", "Mon": "Monday"} #literal notation

capitals = {} #create an empty dictionary then add values

capitals["svk"] = "Bratislava"

capitals["deu"] = "Berlin"

capitals["dnk"] = "Copenhagen"

In the example above, we created two dictionaries in two different ways:

1. Using literal notation. The key-value pairs are enclosed by curly brackets. The pairs are separated by commas. The first value of a pair is a key, which is followed by a colon character and a value. The "Sun" string is a key and the "Sunday" string is a value.
2. Creating empty dictionary and adding some values. The keys are inside the square brackets, the values are located on the right side of the assignment.

**Each key in a dictionary must be unique.**

Using FOR loop to iterate through dictionary:

#to print all keys

for data in capitals:

print data

#another way to print all keys

for key in capitals**.iterkeys()**:

print key

#to print the values

for val in capitals**.itervalues()**:

print val

#to print all keys and values

for key,data in capitals.**iteritems()**:

print key, " = ", data

**Built-in Functions and Methods:**

* **cmp(dict1, dict1)** – compares two dictionaries. The comparison process starts with the length of each dictionary, followed by key names, followed by values. The function returns 0 if the 2 dicts are equal, -1 if dict1 > dict2, 1 if dict1 < dict2.
* **len()** – give the total length of the dictionary
* **str()** – produces a string representation of a dictionary
* **type()** – returns the type of the passed variable. If passed variable is a dictionary, it will then return a dictionary type.
* **.clear()** – removes all elements from the dictionary
* **.copy()** – returns a shallow copy dictionary
* **.fromkeys(sequence, [value])** – create a new dictionary with keys from sequence and values set to value
* **.get(key, default=None)** – For key, returns value or default if key is not in dictionary
* **.has\_key(key)** – returns true if a given key is available in the dictionary, otherwise it returns false
* **.items()** – returns a list of dictionary’s (key, value) tuple pairs
* **.iteritems()** – return an iterator over the dictionary’s (key, value) pairs.
* **.keys()** – return a list of dictionary keys
* **.setdefault(key, default=None)** – similar to get(), but will set dict[key] = default if key is not already in dictionary
* **.update(dict2)** – adds dictionary dict2’s key-value pairs to an existing dictionary
* **.values()** – returns list of dictionary values
* **zip()** – combine two lists like a zipper (i.e. to create a dictionary)
  + superfluous elements won’t be used, whether the extras are keys or values

**Regular Expressions:**

Regular expressions, commonly known as regex, are a set of rules for identifying or matching strings. Regular expression string searches can be what is known as expensive operations. This occurs when general search terms are used, which often force string comparisons to become exponentially complex.

Common uses include searching string inputs from users. Search engines and other form input, like user registration and login, are great examples of correct uses for regular expressions.

|  |  |
| --- | --- |
| . | Matches any character except a new line. |
| \w | Matches any letter or digit. |
| + | The pattern before it can appear 1 or more times. |
| \* | The pattern can appear any number of times, including none. |

**re.search(pattern, string)** is a function that scans a string for a specific regex pattern and returns a match object if it finds a match and None otherwise.

Ex:

if re.search(r"a.\*a"):

print("That string had at least two 'a's in it!")

else:

print("No more than one 'a' found!")