Python Primer: For OO Devs

https://github.com/sean-blessing/python-primer-20260214

GitHub	LI

Instructor: Sean Blessing

Updated: February 2026

Overview & Purpose

Provide an overview of the Python programming language in a brief session, from the perspective of a developer who already understands OO programming. This allows an expedited approach, focusing on syntax differences, high level explanation, in class demonstrations, and hands-on labs demonstrating these differences through simple console projects.

Prerequisites

1. Development experience in another OO language such as Java or C#.
    a. We won't be going in depth to explain core concepts like control structures, methods, classes, OO concepts in general
    b. We will be doing brief, in class demonstrations, followed up by lab time, if time allows
2. Experience with multiple IDEs such that utilizing a new one is a short learning curve
3. Basic understanding of the Windows Operating System

Objectives

1. Learn basic console app development in Python.
2. Understand major differences in Python approach vs C based languages.
3. If time allows, introduction to advanced / convenience functions available in Python which simplify some of the things that normally take a lot more code to achieve! Ex: Data Analysis, File Processing, GUI

Outline

- Python Install
- Python Overview
- Key Differences Overview
- Python / IDE Overview, First Apps, statements, variables, basic control structures (if, while)
- Operators, Math module
- More Control Statements
- Functions, Exceptions
- Debugger
- Classes
- Lists / Dictionaries, looping, finding, processing
- File Processing

Setup

1. Python Install
    a. Download the latest version here. Version 3.14.3 as of February, 2026
    b. Be logged in as an administrator on your laptop
    c. Run the executable
    d. Check the box to add Python to PATH
    e. During install, if you see an option “Disable path length limit”, click it
    f. To confirm install, go to a command prompt and type `python -V` [‘V’ must be uppercase]. This should show you the current installed version.
2. IDE - Most IDEs will have a Python interpreter installed, so you are welcome to use whichever you prefer. The examples in this material use VS Code, but any IDE should work.
    a. Visual Studio Code Download can be found here.

Python Overview / Definition

From Python.org: “Python is an **interpreted**, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with **dynamic typing** and **dynamic binding**, make it very attractive for Rapid Application Development, as well as for use as a **scripting** or glue language to connect existing components together. Python's ***simple, easy to learn syntax*** emphasizes readability and therefore reduces the cost of program maintenance.”

From Wikipedia: “Python is an interpreted high-level **general-purpose programming language.** Python's design philosophy emphasizes code readability with its notable use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.”

Python vs Java and C# - Python.org

- Python programs are generally expected to run slower than Java/C# programs
    - Because of the run-time typing, Python's run time must work harder than Java's.
- Python Programs take much less time to develop

PIP

- PIP is the package manager in Python
- Similar to NuGet in C#, Maven in Java
- Use it when you need a package that's not part of core Python

Sean's Take

- Python is kind of a jack of all trades language… OO, Functional and Scripting
- That sounds really cool, multi-functional, but if you are used to a traditional OO language it can be tough to get used to
- Python provides a lot of 'convenience' functions out-of-the-box… conveniences that we'd otherwise have to write multiple statements to perform the same functionality. See here for a list of Python built-in functions
- It is used A TON in Data Science and AI
- Syntax differences are the biggest hurdle for Java, C# programmers looking to skill-up in Python. As I always say, "Repetition Breeds Cognition"!!!

Biggest Differences (Python vs Java/.Net)

No semicolons!	Colons used at start of control structures	“print” vs System.out…, Console.print...
No curly braces!	“elif” vs “else if”	“True” vs “true”
Indentations mark ‘blocks’	“not” is synonymous w/ “!”	No ++, -- operator?!?!
Dynamic typing, types defined at assignment (think JavaScript)	Can run as script without full class structure	Variable scope is WAY different
Types available	var_name_convention	Dictionary (Map) usage very different
Function definition syntax ‘def’ keyword	Class constructor replaced by __init__() function	IDE diffs… no auto line breaks after function / method definition
None vs null	‘self’ vs ‘this’	No ‘new’ keyword

- Python source code is ‘interpreted’ into bytecode.
- They are stored in a ‘pychache’ folder, but only once we’re utilizing modules

A Lot of Stuff is the Same

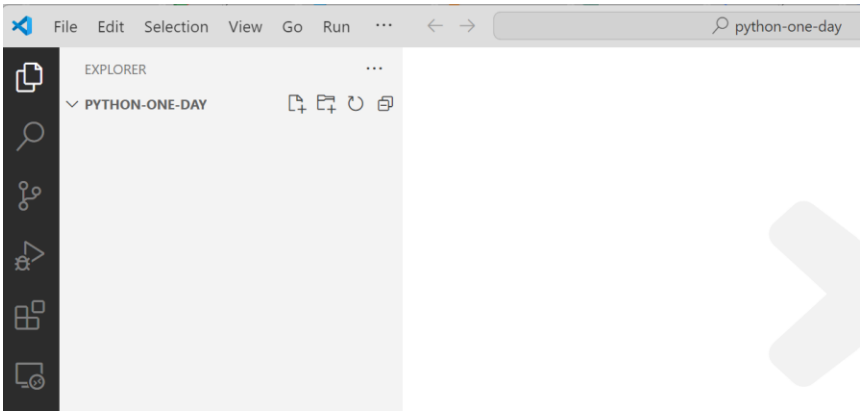
- Top down code execution
- Creating a project
- “if”, “while” statements, “break”, “continue” keywords
- +=, -=, == operators

Similarities to JavaScript

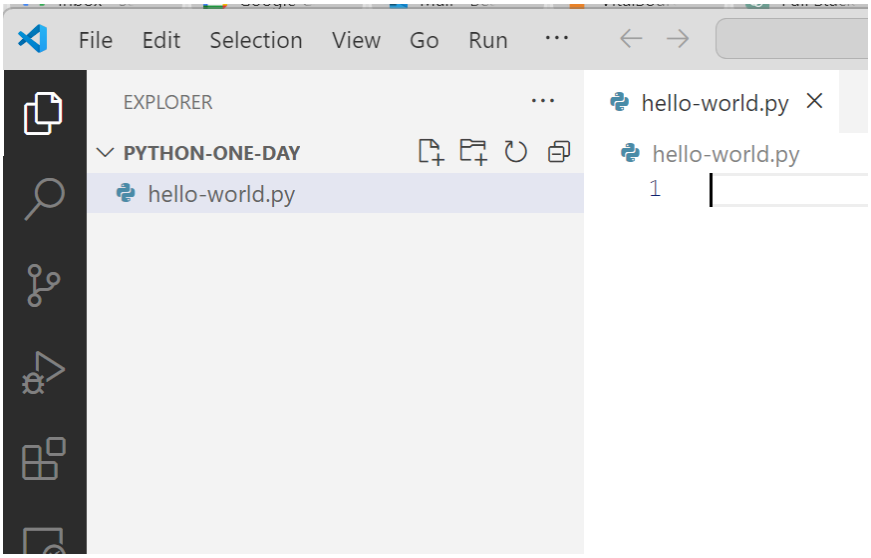
- Dynamic typing
- Single vs double quotes are interchangeable
- “slice” of lists
    - In both Java and C# there is no explicit ‘slice’, but other methods do something similar [ex: copyOfRange(), substring() in Java]

Open VS Code

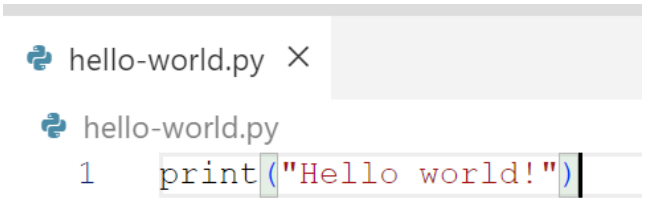
Open a new folder in VS Code, one where we will be writing our Python code.

- 

- Create a new file named `hello-world.py`



- Add the following line to the file:



- Run the file:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PC

PS C:\repos\max\python\python-one-day> &
hon.exe c:/repos/max/python/python-one-da
Hello world!
PS C:\repos\max\python\python-one-day> &
hon.exe c:/repos/max/python/python-one-da
Hello world!
PS C:\repos\max\python\python-one-day>
```

Course Approach

Chapter(s)	Concept
1, 2	Python / IDE Overview, First Apps, statements, variables, basic control structures (if, while)
3	Operators, Math module
4	More Control Statements
5	Functions, Exceptions
6	Debugger
7	Classes
8	Lists / Dictionaries, looping, finding, processing
9	File Processing

Chapter 2 - first applications

- Code completion and syntax error detection
- Statements
    - No semicolon to end the statement!
    - Statements end w/ 'Enter' key
- No identifiers… just define your variables and type gets assigned dynamically
- Comments
    - Only single line comments - utilize #
- Basic data types - w3schools
- Arithmetic operators: +, -, *, /, %
    - Most data types you are used to are available, just named a little differently
- You can utilize a lot of built in functions, demonstrated in class
- Console interaction:
    - No System.out.println(), System.out.print()
    - No Console.Write(), Console.WriteLine()
    - print("..." + "..."): Print whatever you want! You CAN concatenate, but it gets tricky when you introduce variables
    - print(f"....{ variable_name }...") -> similar to interpolation
    - Always ends w/ a new line ("\n").
    - 'end' attribute
        - If you want to end a print without a new line: print("....", end="")
    - Print multiple variables/values w/ a separator
        - print('a', 'b', 'c', sep="-") -> a-b-c
    - Get user input: input("....")
        - String inside parenthesis will be the prompt
        - Always returns a string
        - Many ways to convert strings
            - int(...), float(...)
        - Convert numerics to strings - str(...)

        - Get input from arguments passed to program
            - import sys
              print(sys.argv[0]) -> prints program name
              print(sys.argv[1]) -> prints 1st argument passed
- If statements

```
code = "r"
if code == "r":
    print("Red")
elif code == "b":
    print("Blue")
else:
    print("Other")
```

    - Notes:
        - Curly braces replaced w/ single colon
        - “elif” replaces “else if”
- While loop

```
choice = "y"
while choice != "n":
    print("hello")
    choice = input("Go again?")
```

Labs

- Grade Converter [2-2]
    - New Concepts Demonstrated:
        - print
        - input
        - while loop
        - if statement
- Rectangle Calculator [2-3]
    - New Concepts Demonstrated:
        - Calculations

Chapter 3 - computations, Math module, formatting

Built in functions:

- max / min

```
max_nbr = max(5, 7)      #7
min_nbr = min(8, 3)      #3
```

- Note: Variable naming convention: snake_case, or python_case / snail_case
- sum

```
total = sum([2,3])      #5
total = sum([2,3,5])    #10
```

- Round

```
round_nbr = round(4.567, 2)         #4.57
```

- Python math module

```
import math
ceil_nbr = math.ceil(5.5)       #6
floor_nbr = math.floor(5.5)     #5
```

    - Note: math module must be imported
- Python random module

```
import random
die_roll = random.randint(1,6)      #a random # between 1 and 6, inclusive
```

- String formatting - several options!

```
price = 20000
price_currency = "${:,.2f}".format(price)   #$20,000.00
grade = .9995
grade_pct_1 = format(grade, '%')            #99.950000%
grade_pct_2 = "{:.2%}".format(grade)        #99.95%
print(f"Format % w/ 2 decimals in string: {grade:.2%}") #99.95%
```

Labs

- Temperature Converter [3-1]
- Travel Time Calculator [3-2]

Chapter 4 – arrays & more control statements

- Arrays allow us to store a set of values:

```
# Arrays
even_numbers = [2, 4, 6, 8, 10]
odd_numbers = [1, 3, 7, 9]

# add value at end:
even_numbers.append(12)

# add value at position (index starts at 0):
odd_numbers.insert(2, 5)

print(even_numbers)
print(odd_numbers)
```

- Python doesn't have a `switch` statement, but it does have a `match` statement:

```
command = "list"
match command:
    case "list":
        print("List selected")
    case "add":
        print("Add selected")
    case "edit":
        print("Edit selected")
    case _:                     #default case
        print("Unknown Command. Try again.")
```

- For loop examples - several variations:

```
even_nbrs = [2, 4, 6, 8, 10]
for nbr in even_nbrs:
    print(nbr)
# range starts at 0 (default), stops at 4)
for i in range(len(even_nbrs)):
    print(f"{i}: {even_nbrs[i]}")

for i in enumerate(even_nbrs):
    print(i, i[0], i[1])

for idx, value in enumerate(even_nbrs):
    print(f"{idx}: {value}")

# including 'start' (inclusive) and 'stop' (exclusive)
for n in range(1, 10):
    print(n)

# including 'step'
for n in range(1, 10, 2):
    print(n)
```

- Notes:
  - Square brackets denote a `list` - more on `lists` later.

Labs

- 4-1 Table of Powers
- **4-2 Factorial Calculator**
- 4-3 Tip Calculator

Chapter 5 - functions, exceptions

- Python has both functions and methods
    - In a procedural program / script they are called functions
    - In a class they are called methods
- Use the keyword 'def' to define a function:

```
def add_function(num1, num2):
    return num1 + num2

print(add_function(5, 3))
```

- Assigning default values:

```
def add_function(num1, num2=2):
    return num1 + num2

print(add_function(5))
```

- Variable args (*args):

```
def add_function(*nbrs):
    sum = 0
    for n in nbrs:
        sum += n
    return sum

print(add_function(1, 2))
print(add_function(2, 4, 6))
print(add_function(1, 3, 5, 7, 9))
```

- Keyword args

```
def calc_total_function(price, quantity, handling_fee):
    #(price * quantity) + handling_fee = total
    return (price*quantity) + handling_fee
print(calc_total_function(20, 2, 3))
#43
print(calc_total_function(handling_fee=5, quantity=7, price=10))
#75
```

- Arbitrary Keyword Arguments(**kwargs) -don't know how many args will be passed

```
def my_function(**a_person):
    print ("Name is: " + a_person["l_name"] + ", " +
a_person["f_name"])

my_function(f_name = "Bob", l_name = "Marley")
```

- Access modifiers:
    - Public: The default access, no attribute needed
    - Protected: Single underscore ('_')
    - Private: Double underscore ('__')
- Exceptions - treated similarly as we do in other OO languages but the syntax is just a little different.
- Consider the following code and how it could possibly throw an exception:
    - Notes:
        - `input()` accepts user input from the console
        - `int()` converts a string to a number

```
whole_nbr = int(input("Enter a whole number: "))
print(f"You entered: {whole_nbr}")
```

    - If an alphanumeric value is entered, a ValueError is returned:

```
hi
Enter a whole number: z
Traceback (most recent call last):
  File "C:\repos\sean-self-study\python\python-1-day\ch01\hello-world\main.py", line 3, in <module>
    whole_nbr = int(input("Enter a whole number: "))
ValueError: invalid literal for int() with base 10: 'z'

Process finished with exit code 1
```

    - We can wrap the code in a try/except [Not try/catch!]:

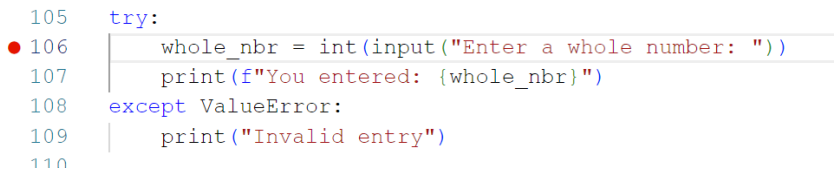
```
try:
    whole_nbr = int(input("Enter a whole number: "))
    print(f"You entered: {whole_nbr}")
except ValueError:
    print("Invalid entry")
```

Labs

- 5-1 Dice Roller
- **5-3 Guessing Game**

Chapter 6 - debugger

- Set a breakpoint by clicking just to the left of the line number in your editor. A red dot will appear.



- The integrated debugger is accessed under the Run -> Start Debuging... menu item.
- Typical debugger options are available (step over, step into, etc.) from the Run menu option.

Chapter 7 - classes, modules

- With Python version 3.7 the `@dataclass` decorator was introduced, which directs Python to generate the `__init__()` method. More on this later.
- Here's an example of a `Person` class, defined in Python v3.7 or later::

```
from dataclasses import dataclass

@dataclass
class Person:
    id: int
    first_name: str
    last_name: str
    email: str
    phone: str
```

  - 'class' declaration - assumed public
  - Class names start with a capital letter.
  - Class names DO NOT have to match the filename.
  - Property names defined as snake_case w/ types
  - No declaration of getter, setters, nor constructor

- A `Person` class defined in Python v6 or earlier:

```
class Person:
    def __init__(self, id = 0.0, first_name = "", last_name = "",
                 email = "", phone = ""):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
```

  - Properties defined inside the `__init__()` method
  - The `self` keyword is used to set properties, similar to `this` in other languages

- A person_manager.py file importing Person, creating an instance, displaying properties, and changing a property:

```
from person import Person

person = Person(1, "Marty", "McFly", "marty@b2f.com", "282-333-1234")
print(f"Person Info:")
print(f"id:     {person.id}")
print(f"name:   {person.first_name} {person.last_name}")
print(f"email:  {person.email}")
print(f"phone:  {person.phone}")
person.phone = "222-333-1234"
print(f"phone:  {person.phone}")
```

    - Notes:
        - Constructing a new instance does NOT require the new keyword
        - Properties are accessed using dot notation (`person.phone`)

- Methods can be defined in a class by using the `def` keyword:

```
   def get_person_details(self):
        details = f"id: {self.id}, first_name: {self.first_name},
last_name: {self.last_name}, "
        details += f"email: {self.email}, phone: {self.phone}"
        return details
```

    - `details` `f string` is split across two lines. Since Python utilizes end of line characters to end statements, we split it into two statements.

Labs

- 7-1 Contact List
    - Contact
        - first_name
        - last_name
        - email
        - Phone
        - print_contact()
    - Prompt user for 4 values of a contact
    - Create an instance of Contact
    - Print the contact info

Chapter 8 – strings as lists, dictionaries, etc.

Every language has various types to manage “lists” of things. Here we will talk about Ptyhon’s list and dictionary types.

- A Python `list` is similar to an `array` in other languages, yet is more dynamic. It is similar to an `ArrayList` in Java or C#. Here are some examples:

```
# A string: chars stored in order
bob_marley_name = "Robert Nesta Marley"

# A list: ordered and changeable, denoted by square brackets
songs = ["Jamming", "Three Little Birds", "No Woman No Cry"]

# A tuple: ordered and NOT changeable, denoted by parenthesis
# Note: this tuple is not complete. Bob Marley had a lot of kids!
children = ("Sharon", "David 'Ziggy'", "Stephen", "Robert 'Robbie'",
"Julian", "Ky-Mani", "Damian")
```

- Processing a string - `bob_marley_name`:

```
#processing a string - bob_marley_name
bob_marley_name = "Robert Nesta Marley"
print('bob_marley_name:', bob_marley_name)
print('length of bob_marley_name', len(bob_marley_name))
#get portions of the bob_marley_name by index and slicing
print('first char:', bob_marley_name[0])
#get first name of bob_marley_name
# where's the first space?
idx_1 = bob_marley_name.index(" ")
f_bob_marley_name = bob_marley_name[0:idx_1]
print(f"First name: {f_bob_marley_name}")
idx_2 = bob_marley_name.index(" ", idx_1+1)
m_bob_marley_name = bob_marley_name[idx_1+1: idx_2]
print(f"Middle name: {m_bob_marley_name}")
l_bob_marley_name = bob_marley_name[idx_2+1:len(bob_marley_name)]
print(f"Last name: {l_bob_marley_name}")
for str in bob_marley_name:
    print(str, end=" ")
```

- Output:

```
bob_marley_name: Robert Nesta Marley
length of bob_marley_name 19
first char: R
First name: Robert
Middle name: Nesta
Last name: Marley
R o b e r t   N e s t a   M a r l e y
```

- Processing a list:

```
# process the songs in a list
# use square brackets and slicing
songs = ["Jamming", "Three Little Birds", "No Woman No Cry"]
print(f"songs:                      {songs}")
print(f"song 1:                     {songs[0]}")
print(f"last song songs[2]:         {songs[2]}")
print(f"last song songs[-1]:        {songs[-1]}")
print(f"last 2 songs songs[1:3]:    {songs[1:3]}")
#append a song to the end
songs.append('temp song')
print(f"temp song added to end:     {songs}")
#change a song/item
songs[3] = "Bobs so cool - not a real song"
print(f"song 3 changed:             {songs}")
# remove the last song and return it
rem_song = songs.pop()
print(f"removed song:               {rem_song}")
print(f"songs:                      {songs}")
```

- Output:

```
songs:                      ['Jamming', 'Three Little Birds', 'No
Woman No Cry']
song 1:                     Jamming
last song songs[2]:         No Woman No Cry
last song songs[-1]:        No Woman No Cry
last 2 songs songs[1:3]:    ['Three Little Birds', 'No Woman No Cry']
temp song added to end:     ['Jamming', 'Three Little Birds', 'No
Woman No Cry', 'temp song']
song 3 changed:             ['Jamming', 'Three Little Birds', 'No
Woman No Cry', 'Bobs so cool - not a real song']
removed song:               Bobs so cool - not a real song
songs:                      ['Jamming', 'Three Little Birds', 'No
Woman No Cry']
```

- Range:

```
#range function
print("\nRanges!")
print(list(range(10)))
print(list(range(5, 17)))
print(list(range(0, 20, 2)))
print(list(range(0, 20, 3)))
print(list(range(0, -10, -1)))
```

- Output:

```
Ranges!
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 3, 6, 9, 12, 15, 18]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

- ‘For .. in’:

```
# for .. in
print("\nfor .. in")
for song in songs:
    print(song)
```

- Output:

```
Ranges!
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 3, 6, 9, 12, 15, 18]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

- ‘for’ using index:

```
# for w/ index
print("\nfor w/ index")
for i in range(0, len(songs)):
    print(f"song [{i}]: {songs[i]}")
```

- Output:

```
for w/ index
song [0]: Jamming
song [1]: Three Little Birds
song [2]: No Woman No Cry
```

- ‘in’ keyword - see if something is in a list:

```
# in keyword - text for existence in a list:
print("'in' keyword - is 2 in the list?")
numbers_list = [1, 2, 1, 3, 1, 4, 1, 5]
if 2 in numbers_list:
    print("yep, it's in there!")
```

- Output:

```
'in' keyword - is 2 in the list?
yep, it's in there!
```

- The '*' operator does more than math when used on strings and lists:

```
# The * operator - more than math multiplier
print("\n* operator")
str = "a" * 10
print(str)
numbers_list = [1, 2, 3]
num_list = numbers_list * 3
print(f"num_list: {num_list}")
```

- Output:

```
* operator
aaaaaaaaaa
num_list: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- Here are some built-in methods for lists
    - A few more: clear(), copy()
- Use random() to get a random item from list:

```
# use random() to get a random item from a list:
print("\nrandom # from list")
import random
numbers_list = [1, 50, 23, 74, 11]
print(f"1st choice: {random.choice(numbers_list)}")
print(f"2nd choice: {random.choice(numbers_list)}")
```

- Output:

```
random # from list
1st choice: 23
2nd choice: 11
```

- Dictionaries

```
# dictionaries
# dictionary key/value can contain any data type
# can access by get method or square brackets
print("\ndictionaries")
spanish_english = {
    "uno": "one",
    "dos": "two",
    "tres": "three",
    1: "one"
}
print(spanish_english)
# get value by key
print(spanish_english.get("dos"))
print(spanish_english[1])
# add elements to a dictionary using square brackets (no add
function)
spanish_english["quatro"] = "four"
print(spanish_english)
```

- Output:

```
dictionaries
{'uno': 'one', 'dos': 'two', 'tres': 'three', 1: 'one'}
two
one
{'uno': 'one', 'dos': 'two', 'tres': 'three', 1: 'one', 'quatro':
'four'}
```

- Iterate through dictionary keys using for...in

```
#iterate over dictionary keys using for .. in
print("\nIterate over dictionary w/ for .. in")
for key in spanish_english:
    print(f"{key} means {spanish_english[key]}")
```

- Output:

```
Iterate over dictionary w/ for .. in
uno means one
dos means two
tres means three
1 means one
quatro means four
```

- .values() returns all values:

```
#.values contains all values
print("\n.values() to return all key-value pairs")
for value in spanish_english.values():
    print(f"{value}", end=" ")
```

- Output:

```
.values() to return all key-value pairs
one two three one four
```

- Get both keys and values using .items():

```
#get both keys and values using .items()
print(".items() returns both key and value")
for k, v in spanish_english.items():
    print(f"k, v: {k}, {v}")
```

- Output:

```
.values() to return all key-value pairs
one two three one four .items() returns both key and value
k, v: uno, one
k, v: dos, two
k, v: tres, three
k, v: 1, one
k, v: quatro, four
```

- Check to see if a key exists in dictionary:

```
#does key exist in dictionary?
print("check key existence...")
if "uno" in spanish_english:
    print("Yes, 'uno' is in dictionary")
```

- Output:

```
check key existence...
Yes, 'uno' is in dictionary
```

- Use pop() to remove last item or pop(key) to remove specific item:

```
#use pop(key) to remove item or popitem() to remove last item
#pop returns removed value, popitem() returns entire item (k,v)
print("\npop and popitem to remove items")
print(spanish_english)
item_1 = spanish_english.pop(1)
print(f"item_1={item_1}")
print(spanish_english)
item_2 = spanish_english.popitem()
print(f"item_2={item_2}")
print(spanish_english)
```

- Output:

```
pop and popitem to remove items
{'uno': 'one', 'dos': 'two', 'tres': 'three', 1: 'one', 'quatro':
'four'}
item_1=one
{'uno': 'one', 'dos': 'two', 'tres': 'three', 'quatro': 'four'}
item_2=('quatro', 'four')
{'uno': 'one', 'dos': 'two', 'tres': 'three'}
```

Chapter 9 - file processing

- Python provides some very simple functions to process files
- Much less overhead than in Java SDK, for sure!

Text File Example

- Given a text file hobbies.txt:

```
music
running
biking
python coding
```

    - File methods:
        - read() - reads whole file as a single long string
        - readline() - reads one line of file into string
        - readlines() - reads file into list of strings
- We can process this file with the following code:

```
print("File Processing")
# read()
print("read() hobbies.txt")
with open('hobbies.txt') as hobbies_file:
    contents = hobbies_file.read()
print(f"read() : {contents}")

# readline()
print("readline() hobbies.txt")
with open('hobbies.txt') as hobbies_file:
    contents = hobbies_file.readline()
print(f"readline() : {contents}")

# readlines()
print("readlines() hobbies.txt")
with open('hobbies.txt') as hobbies_file:
    contents = hobbies_file.readlines()
print(f"readlines() : {contents}")
```

- Output:

```
File Processing
read() hobbies.txt
read() : music
running
biking
python coding
readline() hobbies.txt
readline() : music

readlines() hobbies.txt
readlines() : ['music\n', 'running\n', 'biking\n', 'python coding']
```

- Writing to a file:

```
#writing to a file
#define list of olympic athletes
print("\nWrite to the olympic-athlete file...")
olympic_athletes = ["Simone Biles", "Michael Phelps", "Michael
Johnson", "Chloe Kim"]
#define a file to write to (without the 'with' keyword)
olympic_file = open("olympic-athletes.txt", "w")
#loop through list and write to file
for athlete in olympic_athletes:
    olympic_file.write(f"{athlete}\n")
#close file
olympic_file.close()
print("Done. Check the file.")
```

- Output:

```
Write to the olympic-athlete file...
Done. Check the file.
```

- Check VS Code for the `olympic-athletes.txt` file.

Processing CSV Files

Python provides the `csv` module to make processing CSV files easy.

Writing to a CSV

Let's start with a Movie class, create some movies, and put them in a list:

- The Movie class:

```
from dataclasses import dataclass

@dataclass
class Movie:
    id: int
    title: str
    year: int
    rating: str
    director: str

    def get_movie_details(self):
        details = f"id: {self.id}, title: {self.title}, year:
{self.year}, "
        details += f"rating: {self.rating}, director:
{self.director}"
        return details
```

- The code to use the movie class, create some movies, and write to CSV:

```
#processing CSV Files
print("CSVs")
print("Write to a movies.csv file")
#define some movies
movie_1 = Movie(1, "Star Wars Episode IV: A New Hope", 1977, "PG",
"George Lucas")
movie_2 = Movie(2, "Coco", 2017, "PG", "Lee Unkrich, Adrian Molina")
movie_3 = Movie(3, "Black Panther", 2018, "PG-13", "Ryan Coogler")
#put them in a list
movies = [movie_1, movie_2, movie_3]
#import csv and use it to write the movies to a file using with
keyword
import csv
with open("movies.csv", "w", newline="") as movie_file:
    writer = csv.writer(movie_file)
    for movie in movies:
        writer.writerow([movie.id, movie.title, movie.year,
movie.rating, movie.director])
print("done")
```

- Output

```
CSVs
Write to a movies.csv file
done
```

- Reading a CSV

```
# read the csv
print("\nRead the movie csv and print to console...")
with open("movies.csv", newline="") as movie_file:
    reader = csv.reader(movie_file)
    for movie in reader:
        print(f"Movie: {movie[1]} ({movie[2]}), rated {movie[3]},
directed by {movie[4]}")
print("done")
```

- Output

```
Read the movie csv and print to console...
Movie: Star Wars Episode IV: A New Hope (1977), rated PG, directed by
George Lucas
Movie: Coco (2017), rated PG, directed by Lee Unkrich, Adrian Molina
Movie: Black Panther (2018), rated PG-13, directed by Ryan Coogler
done
```

- Writing from a List of Lists rather than a List of Objects
    - This is a little simpler with the csv library, so it depends on the situation:

```
# writing csv from a list of lists
print("\nWrite a movies csv from a list of lists")
movies_list = [["Star Wars Episode IV: A New Hope", 1976],
    ["Sixteen Candles", 1984],
    ["Rogue One", 2016],
    ["Happy Gilmore", 1996],
    ["Wedding Crashers", 2005]]
# write list to a csv
with open("movies-list.csv", "w", newline="") as movie_file:
    writer = csv.writer(movie_file)
    writer.writerows(movies_list)

#p.219 csv.reader()
with open("movies-list.csv", newline="") as movie_file:
    reader = csv.reader(movie_file)
    for movie in reader:
        print(f"{movie[0]} ({movie[1]})")
print("done")
```

- Output

```
Write a movies csv from a list of lists
Star Wars Episode IV: A New Hope (1976)
Sixteen Candles (1984)
Rogue One (2016)
Happy Gilmore (1996)
Wedding Crashers (2005)
done
```

Chapter 10 - date and time

- Python provides several modules to process time, datetime, etc.
- W3schools datetime
- Basic Date Stuff::

```
# basic date stuff
print("Basic date definition...")
date_today = date.today()
print(f"date_today: {date_today}")
datetime_today = datetime.today()
print(f"date_today: {datetime_today}")
datetime_now = datetime.now()
print(f"date_now: {datetime_now}")
time_now = datetime_now.time()
print(f"time_now: {time_now}")

christmas = date(2024,12,25)
print(f"christmas: {christmas}")
time_eleven = time(11,11,11)
print(f"time_eleven: {time_eleven}")
```

- Output

```
Basic date definition...
date_today: 2024-05-16
date_today: 2024-05-16 11:30:17.286177
date_now: 2024-05-16 11:30:17.286176
time_now: 11:30:17.286176
christmas: 2024-12-25
time_eleven: 11:11:11
```

- Datetime parsing strings

```
#p.307 datetime parsing strings
print("\nParsing datetime strings:")
christmas_date_str = "12/25/24"
halloween_date_str = "2024-10-31"
appointment_date_time_str = "2024-05-09 08:00"

christmas_date = datetime.strptime(christmas_date_str, '%m/%d/%y')
print(f"christmas_date: {christmas_date}")
halloween_date = datetime.strptime(halloween_date_str, '%Y-%m-%d')
print(f"halloween_date: {halloween_date}")
appointment_date = datetime.strptime(appointment_date_time_str, '%Y-
%m-%d %H:%M')
print(f"appointment_date: {appointment_date}")
```

- Output

```
Parsing datetime strings:
christmas_date: 2024-12-25 00:00:00
halloween_date: 2024-10-31 00:00:00
appointment_date: 2024-05-09 08:00:00
```

- Formatting Dates

```
# #p.309 - formatting dates
print("\nFormatting dates:")
christmas_date_formatted = christmas_date.strftime("%m/%d/%y")
print(f"christmas_date_formatted: {christmas_date_formatted}")
halloween_date_formatted = halloween_date.strftime("%Y-%m-%d")
print(f"halloween_date_formatted: {halloween_date_formatted}")
appointment_date_formatted = appointment_date.strftime("%Y-%m-%d
%H:%M")
print(f"appointment_date_formatted: {appointment_date_formatted}")

# another way to format a date...
halloween_datetime = datetime(1988, 10, 31, 22, 48)
print(f"halloween_datetime: {halloween_datetime}")
halloween_date_formatted = f"{halloween_datetime:%Y-%m-%d}"
print(f"halloween_datetime_formatted(v2):
{halloween_date_formatted}")
```

- Output

```
Formatting dates:
christmas_date_formatted: 12/25/24
halloween_date_formatted: 2024-10-31
appointment_date_formatted: 2024-05-09 08:00
halloween_datetime: 1988-10-31 22:48:00
halloween_datetime_formatted(v2): 1988-10-31
```

- Comparing Dates - deltas

```
# spans of time
from datetime import timedelta
print("--- timedeltas ---")
three_weeks = timedelta(weeks=3)
print(f"today = {date_today}")
print(f"Three weeks from today: {date_today + three_weeks}")
print(f"Three weeks ago today: {date_today - three_weeks}")
time_span_until_christmas = christmas - date_today
print(f"Time Span until christmas: {time_span_until_christmas}")
print(f"Days until christmas: {time_span_until_christmas.days}")
```

- Output

```
--- timedeltas ---
today = 2024-05-16
Three weeks from today: 2024-06-06
Three weeks ago today: 2024-04-25
Time Span until christmas: 223 days, 0:00:00
Days until christmas: 223
```

Chapter 11 - JSON module

- JSON is similar to a dictionary
- Import json module
- **json.load**: convert JSON to Python dictionary
- **json.dump**: convert Python to JSON
- VS Code: format JSON to readable format - **right-click** -> **Format Document**
- Load Example:
    - JSON file - people.json

```
{
  "people": [
    {
      "name": "Mickey Mouse",
      "email": "mickey@disney.com"
    },
    {
      "name": "Stitch",
      "email": "stitch@disney.com"
    },
    {
      "name": "Tiana",
      "email": "tiana@disney.com"
    }
  ]
}
```

    - Python code utilizing **people.json**:

```
import json

#working with json - load
print("working with json")
print("read people json and print each person...")
with open('people.json') as json_file:
    data = json.load(json_file)
    print(f"type returned from json.load: {type(data)}")
    for person in data['people']:
        print(f"Person: name={person['name']},
email={person.get('email')}")
```

- Output:

```
working with json
read people json and print each person...
type returned from json.load: <class 'dict'>
Person: name=Mickey Mouse, email=mickey@disney.com
Person: name=Stitch, email=stitch@disney.com
Person: name=Tiana, email=tiana@disney.com
```

- Dump example:
  - Python file:

```
#working with json - dump
print("\njson.dump")
data = {}
data['actors'] = []
data['actors'].append({
    'name': 'Mark Hamill', 'email': 'luke@starwars.com'
})
data['actors'].append({
    'name': 'Dwayne Johnson', 'email': 'rock@therock.com'
})

with open('actors.json', "w") as actor_file:
    json.dump(data, actor_file)

with open('actors-formatted.json', "w") as actor_file:
    json.dump(data, actor_file, indent=4)
print("done - check files")
```

- Running this file produces 2 actors json files:
  - actors.json - data all on one line

```
{"actors": [{"name": "Mark Hamill", "email": "luke@starwars.com"},
{"name": "Dwayne Johnson", "email": "rock@therock.com"}]}
```

  - Actors-formatted.json - formatted as readable json

```
{
    "actors": [
        {
            "name": "Mark Hamill",
            "email": "luke@starwars.com"
        },
        {
            "name": "Dwayne Johnson",
            "email": "rock@therock.com"
        }
    ]
}
```

Jupyter Notebooks and Use in Data Science

Other Stuff

- Requests module
- Jupyter Notebooks