Introduction to Python

Yulia Newton, Ph.D.

CS152, Introduction to Artificial Intelligence

San Jose State University

Spring 2022

Python programming language

- “Python is an interpreted high-level general-purpose programming language. Its design philosophy emphasizes code readability with its use of significant indentation” - Wikipedia

- Designed by Guido van Rossum in 1980’s and first appeared in use in 1991
  - Current version is Python3, a major language revision from the previous version 2.7 and is not fully backward-compatible

- Multi-paradigm language: object-oriented, procedural, functional, structured, reflective

- https://www.python.org/about/gettingstarted/
- https://www.python.org/downloads/

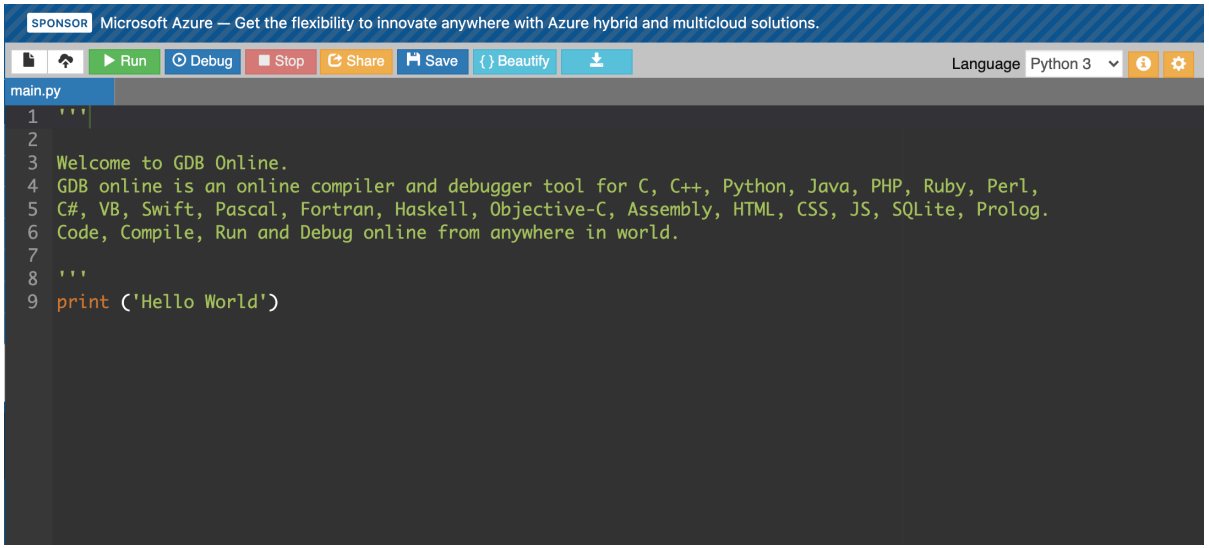
How to run Python

- If you already have a preferred way to execute Python code, then skip this and next slide
- JupyterLab or JupyterNotebooks in Anaconda
  - https://www.anaconda.com/products/individual
- PyCharm is a great IDE for writing and executing Python code
  - https://www.jetbrains.com/help/pycharm/installation-guide.html

  - python indent sensitive

How to run Python (cont'd)

- Free online interpreter at https://www.onlinegdb.com/



Indents matter

- Python is indent sensitive
- Cannot mix indents (tab vs. spaces)
- The first line cannot be indented

Saving code to files

- Python code is saved to files with extension **.py**

Variables

- Python is a dynamically typed language, no type of a variable is declared when the variable is created

```
x = 10
y = 20
print(x + y)

p = "Hello"
q = "World"
print(p + " " + q)
```

https://beginnersbook.com

Multiple variable assignment

```
x = y = z = 99
print(x)
print(y)
print(z)
```

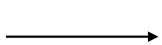
```
99
99
99
```

```
a, b, c = 5, 6, 7
print(a)
print(b)
print(c)
```

```
5
6
7
```

Mathematical operations and concatenations

```
x = 10
y = 20
print(x + y)

p = "Hello"
q = "World"
print(p + " " + q)
```



```
30
Hello World
```

https://beginnersbook.com

Numeric types

- The following numeric types are available in Python: integer, float, complex number

```
# int
num1 = 10
num2 = 100
print(num1+num2)

# float
a = 10.5
b = 8.9
print(a-b)

# complex numbers
x = 3 + 4j
y = 9 + 8j
print(y-x)
```



```
110
1.5999999999999996
(6+4j)
```

https://beginnersbook.com

type() function

```
# int
num = 100
print("type of num: ",type(num))

# float
num2 = 10.99
print("type of num2: ",type(num2))

# complex numbers
num3 = 3 + 4j
print("type of num3: ",type(num3))
```



```
type of num:  <class 'int'>
type of num2:  <class 'float'>
type of num3:  <class 'complex'>
```

isinstance() function

```
num = 100
# true because num is an integer
print(isinstance(num, int))

# false because num is not a float
print(isinstance(num, float))

# false because num is not a complex number
print(isinstance(num, complex))
```



```
True
False
False
```

Integer vs. floating point division

- In Python2.7, dividing two integer values will produce an integer value
  - To obtain a floating point result, at least one operant must be converted to float
- In Python3, dividing two integer values will produce a floating point value
  - In order to perform integer division, we must use // operator

```
print(3 / 2)  # prints 1.5

print(3 // 2)  # prints 1
```

https://discuss.codecademy.com

Immutable data types

- The following are immutable data types in Python:
  - Numeric types
    - We just went through some examples
  - String
  - Tuple

Working with String variables

```
str = 'beginnersbook'
print(str)

str2 = "Chaitanya"
print(str2)

# multi-line string
str3 = """Welcome to
   Beginnersbook.com"""
print(str3)

str4 = '''This is a tech
   blog'''
print(str4)
```



```
beginnersbook
Chaitanya
Welcome to
   Beginnersbook.com
This is a tech
   blog
```

Working with String variables (cont'd)

```
str = "Kevin"

# displaying whole string
print(str)

# displaying first character of string
print(str[0])

# displaying third character of string
print(str[2])

# displaying the last character of the string
print(str[-1])

# displaying the second last char of string
print(str[-2])
```



```
Kevin
K
v
n
i
```

Working with String variables (cont'd)

```
str = "Beginnersbook"

# displaying whole string
print("The original string is: ", str)

# slicing 10th to the last character
print("str[9:]: ", str[9:])

# slicing 3rd to 6th character
print("str[2:6]: ", str[2:6])

# slicing from start to the 9th character
print("str[:9]: ", str[:9])

# slicing from 10th to second last character
print("str[9:-1]: ", str[9:-1])
```



```
The original string is:  Beginnersbook
str[9:]:  book
str[2:6]:  ginn
str[:9]:  Beginners
str[9:-1]:  boo
```

Working with String variables (cont'd)

```
str = "ABC"

# repeating the string str by 3 times
print(str*3)
```

Working with String variables (cont'd)

```
str = "Welcome to beginnersbook.com"
str2 = "Welcome"
str3 = "Chaitanya"
str4 = "XYZ"

# str2 is in str? True
print(str2 in str)

# str3 is in str? False
print(str3 in str)

# str4 not in str? True
print(str4 not in str)
```



```
True
False
True
```

Python is a strongly typed language (cannot do implicit operations on strings and numeric types)

```
s = "one"
n = 2
print(s+n)
```



```
TypeError: must be str, not int
```

Tuples in Python

- A tuple is a collection of elements
  - Similar to list but immutable
  - Elements in a tuple are enclosed in ()
  - E.g. (1,2,3)
  - E.g. (elem1, elem2)
  - E.g. (var1, var2, var3, var4)

Working with tuples

```
# tuple of strings
my_data = ("hi", "hello", "bye")
print(my_data)

# tuple of int, float, string
my_data2 = (1, 2.8, "Hello World")
print(my_data2)

# tuple of string and list
my_data3 = ("Book", [1, 2, 3])
print(my_data3)

# tuples inside another tuple
# nested tuple
my_data4 = ((2, 3, 4), (1, 2, "hi"))
print(my_data4)
```



```
('hi', 'hello', 'bye')
(1, 2.8, 'Hello World')
('Book', [1, 2, 3])
((2, 3, 4), (1, 2, 'hi'))
```

Working with tuples (cont'd)

Empty tuple:

```
my_data = ()
```

Tuple with a single element:

```
my_data = (99,)
```

Note: without a comma, Python will treat my_data as numeric type

Working with tuples (cont'd)

```
# tuple of strings
my_data = ("hi", "hello", "bye")

# displaying all elements
print(my_data)

# accessing first element
# prints "hi"
print(my_data[0])

# accessing third element
# prints "bye"
print(my_data[2])
```



```
('hi', 'hello', 'bye')
hi
bye
```

```
my_data = (1, 2, "Kevin", 8.9)

# accessing last element
# prints 8.9
print(my_data[-1])

# prints 2
print(my_data[-3])
```



```
8.9
2
```

Working with tuples (cont'd)

```
my_data = (1, "Steve", (11, 22, 33))

# prints 'v'
print(my_data[1][3])

# prints 22
print(my_data[2][1])
```



```
v
22
```

Working with tuples (cont'd)

```
my_data = (11, 22, 33, 44, 55, 66, 77, 88, 99)
print(my_data)

# elements from 3rd to 5th
# prints (33, 44, 55)
print(my_data[2:5])

# elements from start to 4th
# prints (11, 22, 33, 44)
print(my_data[:4])

# elements from 5th to end
# prints (55, 66, 77, 88, 99)
print(my_data[4:])

# elements from 5th to second last
# prints (55, 66, 77, 88)
print(my_data[4:-1])

# displaying entire tuple
print(my_data[:])
```



```
(11, 22, 33, 44, 55, 66, 77, 88, 99)
(33, 44, 55)
(11, 22, 33, 44)
(55, 66, 77, 88, 99)
(55, 66, 77, 88)
(11, 22, 33, 44, 55, 66, 77, 88, 99)
```

Working with tuples (cont'd)

```
my_data = (11, 22, 33, 44, 55, 66, 77, 88, 99)
print(my_data)

# true
print(22 in my_data)

# false
print(2 in my_data)

# false
print(88 not in my_data)

# true
print(101 not in my_data)
```



```
(11, 22, 33, 44, 55, 66, 77, 88, 99)
True
False
False
True
```

Working with tuples (cont'd)

```
# tuple of fruits
my_tuple = ("Apple", "Orange", "Grapes", "Banana")

# iterating over tuple elements
for fruit in my_tuple:
    print(fruit)
```



```
Apple
Orange
Grapes
Banana
```

Working with tuples (cont'd)

- Modifying or removing elements of a tuple is illegal

Mutable data types

- The following are mutable data types in Python:
  - List
  - Dictionary
  - Set

Working with a list

- A list is a collection of elements
- Unlike a tuple, list is mutable and elements of a list can be modified or removed
- Elements in a list are enclosed in []
  - E.g. [1,2,3]
  - E.g. [elem1, elem2]
  - E.g. [var1, var2, var3, var4]

```
# list of floats
num_list = [11.22, 9.9, 78.34, 12.0]

# list of int, float and strings
mix_list = [1.13, 2, 5, "beginnersbook", 100, "hi"]

# an empty list
nodata_list = []
```

https://beginnersbook.com

Working with a list (cont'd)

```
# a list of numbers
numbers = [11, 22, 33, 100, 200, 300]

# prints 11
print(numbers[0])

# prints 300
print(numbers[5])

# prints 22
print(numbers[1])
```



```
11
300
22
```

Working with a list (cont'd)

- Python is a strongly typed language and once the types of the list elements has been established, you cannot modify the elements to a different type

```
# a list of numbers
numbers = [11, 22, 33, 100, 200, 300]

# error
print(numbers[1.0])
```

→

```
TypeError: list indices must be integers or slices, not float
```

Working with a list (cont'd)

```
# a list of strings
my_list = ["hello", "world", "hi", "bye"]

# prints "bye"
print(my_list[-1])

# prints "world"
print(my_list[-3])

# prints "hello"
print(my_list[-4])
```



```
bye
world
hello
```

Working with a list (cont'd)

```
# list of numbers
n_list = [1, 2, 3, 4, 5, 6, 7]

# list items from 2nd to 3rd
print(n_list[1:3])

# list items from beginning to 3rd
print(n_list[:3])

# list items from 4th to end of list
print(n_list[3:])

# Whole list
print(n_list[:])
```



```
[2, 3]
[1, 2, 3]
[4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
```

Working with a list (cont'd)

```
# list of numbers
n_list = [1, 2, 3, 4]

# 1. adding item at the desired location
# adding element 100 at the fourth location
n_list.insert(3, 100)

# list: [1, 2, 3, 100, 4]
print(n_list)

# 2. adding element at the end of the list
n_list.append(99)

# list: [1, 2, 3, 100, 4, 99]
print(n_list)

# 3. adding several elements at the end of list
# the following statement can also be written like this:
# n_list + [11, 22]
n_list.extend([11, 22])

# list: [1, 2, 3, 100, 4, 99, 11, 22]
print(n_list)
```

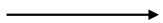


```
[1, 2, 3, 100, 4]
[1, 2, 3, 100, 4, 99]
[1, 2, 3, 100, 4, 99, 11, 22]
```

Working with a list (cont'd)

- Lists are mutable data type, therefore we can modify elements of a list

```
# list of numbers
n_list = [1, 2, 3, 4]

# Changing the value of 3rd item
n_list[2] = 100

# list: [1, 2, 100, 4]
print(n_list)

# Changing the values of 2nd to fourth items
n_list[1:4] = [11, 22, 33]

# list: [1, 11, 22, 33]
print(n_list)
```



```
[1, 2, 100, 4]
[1, 11, 22, 33]
```

Working with a list (cont'd)

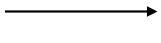
- Lists are mutable data type, therefore we can remove elements of a list

```
# list of numbers
n_list = [1, 2, 3, 4, 5, 6]

# Deleting 2nd element
del n_list[1]

# list: [1, 3, 4, 5, 6]
print(n_list)

# Deleting elements from 3rd to 4th
del n_list[2:4]

# list: [1, 3, 6]
print(n_list)

# Deleting the whole list
del n_list
```



```
[1, 3, 4, 5, 6]
[1, 3, 6]
```

```
# list of chars
ch_list = ['A', 'F', 'B', 'Z', 'O', 'L']

# Deleting the element with value 'B'
ch_list.remove('B')

# list: ['A', 'F', 'Z', 'O', 'L']
print(ch_list)

# Deleting 2nd element
ch_list.pop(1)

# list: ['A', 'Z', 'O', 'L']
print(ch_list)

# Deleting all the elements
ch_list.clear()

# list: []
print(ch_list)
```

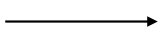


```
['A', 'F', 'Z', 'O', 'L']
['A', 'Z', 'O', 'L']
[]
```

Working with a list (cont'd)

- Although a tuple is an immutable type, if an element of a tuple is a list then the elements of that list can be changed

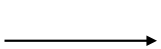
```
my_data = (1, [9, 8, 7], "World")
print(my_data)

# changing the element of the list
# this is valid because list is mutable
my_data[1][2] = 99
print(my_data)
```



```
(1, [9, 8, 7], 'World')
(1, [9, 8, 99], 'World')
```

Working with a list (cont’d)

```
h_letters = []

for letter in 'human':
    h_letters.append(letter)

print(h_letters)
```

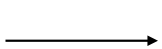


```
['h', 'u', 'm', 'a', 'n']
```

Working with a list (cont'd)

- List comprehension is about manipulating lists into other lists
- List comprehension is an example of declarative programming paradigm in Python
  - Functional programming
- We will cover Python list comprehension when we talk about declarative programming paradigms
  - Right now we are looking at Python from imperative programming point of view

```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
```



```
['h', 'u', 'm', 'a', 'n']
```

Working with dictionaries

- A dictionary is a key-value pair data type
  - Similar to Map in Java
- Contains unique keys associated with values
  - Values do not have to be unique
  - Values can be complex types, e.g. list, tuple, another dictionary, etc.
- Elements of a dictionary are enclosed in {}
  - Value is separated from a key by :
- Values can be accessed by the key
  - If the specified key does not exist in the dictionary then an error is returned

```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
```

Working with dictionaries (cont'd)

```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
print("Student Age is:", mydict['StuAge'])
print("Student City is:", mydict['StuCity'])
```



```
Student Age is: 30
Student City is: Agra
```

Working with dictionaries (cont'd)

- A dictionary is a mutable type, so elements of a dictionary can be modified

```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
print("Student Age before update is:", mydict['StuAge'])
print("Student City before update is:", mydict['StuCity'])
mydict['StuAge'] = 31
mydict['StuCity'] = 'Noida'
print("Student Age after update is:", mydict['StuAge'])
print("Student City after update is:", mydict['StuCity'])
```



```
Student Age before update is: 30
Student City before update is: Agra
Student Age after update is: 31
Student City after update is: Noida
```

Working with dictionaries (cont'd)

- A dictionary is a mutable type, so new elements can be added to a dictionary by specifying a new key

```
mydict = {'StuName': 'Steve', 'StuAge': 4, 'StuCity': 'Agra'}
mydict['StuClass'] = 'Jr.KG'
print("Student Name is:", mydict['StuName'])
print("Student Class is:", mydict['StuClass'])
```



```
Student Name is: Steve
Student Class is: Jr.KG
```

Working with dictionaries (cont'd)

```
mydict = {'StuName': 'Steve', 'StuAge': 4, 'StuCity': 'Agra'}
for e in mydict:
  print("Key:",e,"Value:",mydict[e])
```



```
Key: StuName Value: Steve
Key: StuAge Value: 4
Key: StuCity Value: Agra
```

Alternatively, could say *mydict.keys()*

Working with dictionaries (cont'd)

- A dictionary is a mutable type, so elements of a dictionary can be removed

```
mydict = {'StuName': 'Steve', 'StuAge': 4, 'StuCity': 'Agra'}
del mydict['StuCity']; # remove entry with key 'StuCity'
mydict.clear();     # remove all key-value pairs from mydict
del mydict ;        # delete entire dictionary mydict
```

https://beginnersbook.com

Working with sets

- A set is a collection of elements but it is different than a list - it contains unique elements only
- Elements of a set are enclosed in {}
  - Unlike a dictionary, the elements are not pairs
- While a set is mutable, the elements of a set must be immutable

Working with sets (cont'd)

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```



```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

Working with sets (cont'd)

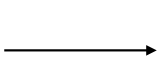
```
# set cannot have duplicates
# Output: {1, 2, 3, 4}
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)

# we can make set from a list
# Output: {1, 2, 3}
my_set = set([1, 2, 3, 2])
print(my_set)

# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.

my_set = {1, 2, [3, 4]}
```



```
{1, 2, 3, 4}
{1, 2, 3}
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    my_set = {1, 2, [3, 4]}
TypeError: unhashable type: 'list'
```

Working with sets (cont'd)

```
# Distinguish set and dictionary while creating empty set

# initialize a with {}
a = {}

# check data type of a
print(type(a))

# initialize a with set()
a = set()

# check data type of a
print(type(a))
```



```
<class 'dict'>
<class 'set'>
```

Working with sets (cont'd)

- A set is mutable type and therefore can be modified by adding elements

```
# initialize my_set
my_set = {1, 3}
print(my_set)

# my_set[0]
# if you uncomment the above line
# you will get an error
# TypeError: 'set' object does not support indexing

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
print(my_set)
```



```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

Working with sets (cont'd)

- A set is mutable type and therefore can be modified by removing elements

```
# Difference between discard() and remove()

# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# you will get an error.
# Output: KeyError

my_set.remove(2)
```



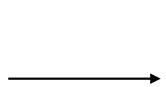
```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line 28, in <module>
KeyError: 2
```

programiz.com

Working with sets (cont'd)

- Can use pop() and clear() functions as well

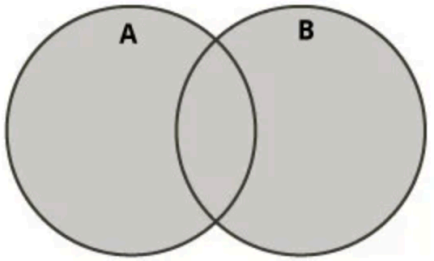
```
# initialize my_set
# Output: set of unique elements
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())

# pop another element
my_set.pop()
print(my_set)

# clear my_set
# Output: set()
my_set.clear()
print(my_set)

print(my_set)
```



```
{'H', 'l', 'r', 'W', 'o', 'd', 'e'}
H
{'r', 'W', 'o', 'd', 'e'}
set()
```

Working with sets (cont'd)

- Union set operation




```
# Set union method
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
print(A | B)
```

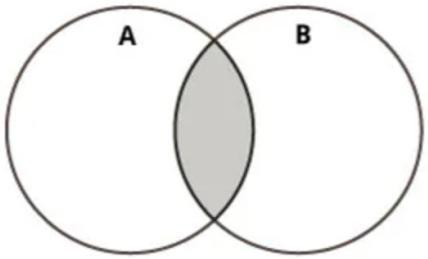
→

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

```
# use union function
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Working with sets (cont'd)

- Intersection set operation



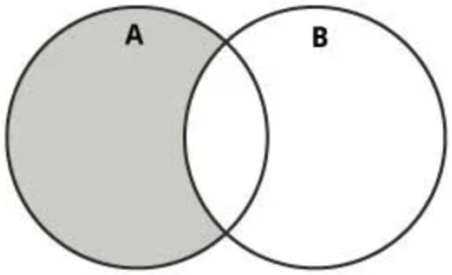

```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
# Output: {4, 5}
print(A & B)
```

```
{4, 5}
```

```
# use intersection function on A
>>> A.intersection(B)
{4, 5}

# use intersection function on B
>>> B.intersection(A)
{4, 5}
```

Working with sets (cont'd)

- Difference set operation (order matters)



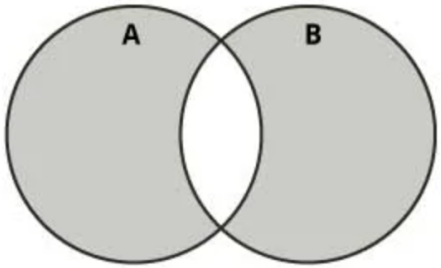

```
# Difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use - operator on A
# Output: {1, 2, 3}
print(A - B)
```

→ `{1, 2, 3}`

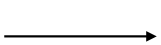
```
# use difference function on A
>>> A.difference(B)
{1, 2, 3}

# use - operator on B
>>> B - A
{8, 6, 7}

# use difference function on B
>>> B.difference(A)
{8, 6, 7}
```

Working with sets (cont'd)

- Symmetric difference set operation (order does not matter)




```
# Symmetric difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator
# Output: {1, 2, 3, 6, 7, 8}
print(A ^ B)
```



```
{1, 2, 3, 6, 7, 8}
```

```
# use symmetric_difference function on A
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}

# use symmetric_difference function on B
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Control flow in Python

- If-else
- Switch
- For loop
- While loop
- Break, continue, pass

If-else

Condition operators:

Equals: `a == b`
Not Equals: `a != b`
Less than: `a < b`
Less than or equal to: `a <= b`
Greater than: `a > b`
Greater than or equal to: `a >= b`

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Inline:

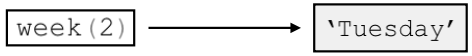
```
a = 2
b = 330
print("A") if a > b else print("B")
```

Inline:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

Switch

```
def week(i):
    switcher={
            0:'Sunday',
            1:'Monday',
            2:'Tuesday',
            3:'Wednesday',
            4:'Thursday',
            5:'Friday',
            6:'Saturday'
        }
    return switcher.get(i,"Invalid day of week")
```



For loop

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Range() function returns a sequence 0 - 5:

```
for x in range(6):
  print(x)
```

```
for x in range(2, 30, 3):
  print(x)
```



```
2
5
8
11
14
17
20
23
26
29
```

While loop

```
i = 1
while i < 6:
  print(i)
  i += 1
```

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Break, continue, pass loop

In place of empty instructions:

```
a = 33
b = 200

if b > a:
  pass
```

Break the loop:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

Continue to the next iteration:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

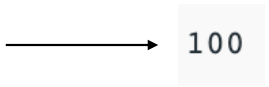
Functions in Python

```
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet('Paul')
```

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num


print(absolute_value(2))

print(absolute_value(-4))
```

Example of a procedural program in Python

```
# Procedural way of finding sum
# of a list

mylist = [10, 20, 30, 40]

# modularization is done by
# functional approach
def sum_the_list(mylist):
    res = 0
    for val in mylist:
        res += val
    return res

print(sum_the_list(mylist))
```



Object oriented programming with Python

- Python can be used to write code that follows object-oriented programming paradigm
- A developer can define classes with attributes/fields and methods

Defining a class in Python

```
class MyClass:
  x = 5
p1 = MyClass()
print(p1.x)
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

→

```
John
36
```

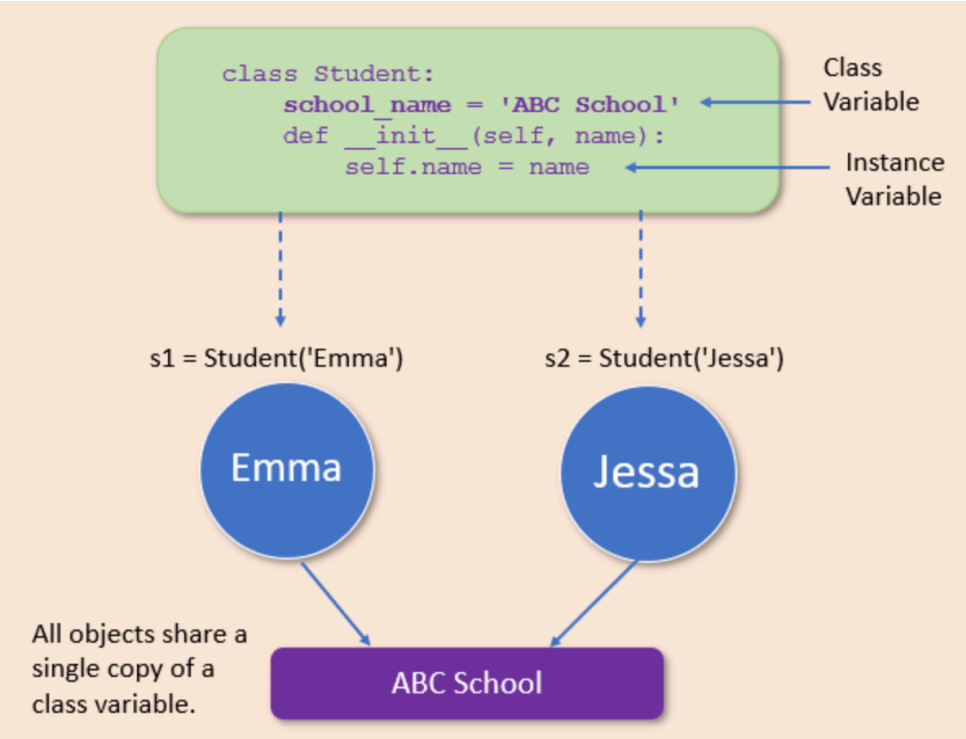
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

→ `Hello my name is John`

Inheritance:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Class vs. instance variables in Python



https://pynative.com/

```python
class Student:
    # Class variable
    school_name = 'ABC School '

    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

# create first object
s1 = Student('Emma', 10)
print(s1.name, s1.roll_no, Student.school_name)
# access class variable

# create second object
s2 = Student('Jessa', 20)
# access class variable
print(s2.name, s2.roll_no, Student.school_name)
```

↓

```
Emma 10 ABC School
Jessa 20 ABC School
```

Initializing instance variables via a constructor

```
class CoffeeOrder:
    def __init__(self, coffee_name, price):
        self.coffee_name = coffee_name
        self.price = price

customer_order = CoffeeOrder("Espresso", 2.10)
print(customer_order.coffee_name)
print(customer_order.price)
```



```
Espresso
2.10
```

Example of object-oriented program in Python

```
# class Emp has been defined here
class Emp:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print("Hello, % s. You are % s old." % (self.name, self.age))

# Objects of class Emp has been
# made here
Emps = [Emp("John", 43),
    Emp("Hilbert", 16),
    Emp("Alice", 30)]

# Objects of class Emp has been
# used here
for emp in Emps:
    emp.info()
```



```
Hello, John. You are 43 old.
Hello, Hilbert. You are 16 old.
Hello, Alice. You are 30 old.
```

Accessing class and instance variables

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    # Instance method
    def show(self):
        print('Inside instance method')
        # access using self
        print(self.name, self.roll_no, self.school_name)
        # access using class name
        print(Student.school_name)

# create Object
s1 = Student('Emma', 10)
s1.show()

print('Outside class')
# access class variable outside class
# access using object reference
print(s1.school_name)

# access using class name
print(Student.school_name)
```



```
Inside instance method
Emma 10 ABC School
ABC School

Outside class
ABC School
ABC School
```

Accessing class and instance variables with getattr()

```
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

# Use getattr instead of stud.name
print('Name:', getattr(stud, 'name'))
print('Age:', getattr(stud, 'age'))
```



```
Name: Jessa
Age: 20
```

Modifying class variables

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    # Instance method
    def show(self):
        print(self.name, self.roll_no, Student.school_name)

# create Object
s1 = Student('Emma', 10)
print('Before')
s1.show()

# Modify class variable
Student.school_name = 'XYZ School'
print('After')
s1.show()
```



```
Before
Emma 10 ABC School

After
Emma 10 XYZ School
```

Modifying instance variables

```python
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# modify instance variable
stud.name = 'Emma'
stud.age = 15

print('After')
print('Name:', stud.name, 'Age:', stud.age)
```



```
Before
Name: Jessa Age: 20

After
Name: Emma Age: 15
```

Dynamically adding an instance variable

```
class Student:
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# add new instance variable 'marks' to stud
stud.marks = 75
print('After')
print('Name:', stud.name, 'Age:', stud.age, 'Marks:', stud.marks)
```



```
Before
Name: Jessa Age: 20

After
Name: Jessa Age: 20 Marks: 75
```

Dynamically deleting an instance variable

```python
class Student:
    def __init__(self, roll_no, name):
        # Instance variable
        self.roll_no = roll_no
        self.name = name

# create object
s1 = Student(10, 'Jessa')
print(s1.roll_no, s1.name)

# del name
del s1.name
# Try to access name variable
print(s1.name)
```



```
10 Jessa
AttributeError: 'Student' object has no attribute 'name'
```

Class methods

```python
from datetime import date

class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def calculate_age(cls, name, birth_year):
        # calculate age an set it as a age
        # return new object
        return cls(name, date.today().year - birth_year)

    def show(self):
        print(self.name + "'s age is: " + str(self.age))

jessa = Student('Jessa', 20)
jessa.show()

# create new object using the factory method
joy = Student.calculate_age("Joy", 1995)
joy.show()
```



```
Jessa's age is: 20
Joy's age is: 27
```

Instance methods

```python
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

    # instance method access instance variable
    def show(self):
        print('Name:', self.name, 'Age:', self.age)

# create first object
print('First Student')
emma = Student("Jessa", 14)
# call instance method
emma.show()

# create second object
print('Second Student')
kelly = Student("Kelly", 16)
# call instance method
kelly.show()
```



```
First Student
Name: Jessa Age: 14

Second Student
Name: Kelly Age: 16
```

Setter methods

```python
class Student:
    def __init__(self, roll_no, name, age):
        # Instance variable
        self.roll_no = roll_no
        self.name = name
        self.age = age

    # instance method access instance variable
    def show(self):
        print('Roll Number:', self.roll_no, 'Name:', self.name, 'Age:', self.age)

    # instance method to modify instance variable
    def update(self, roll_number, age):
        self.roll_no = roll_number
        self.age = age

# create object
print('class VIII')
stud = Student(20, "Emma", 14)
# call instance method
stud.show()

# Modify instance variables
print('class IX')
stud.update(35, 15)
stud.show()
```



```
class VIII
Roll Number: 20 Name: Emma Age: 14
class IX
Roll Number: 35 Name: Emma Age: 15
```

Procedural vs. OO program in Python

Procedural program:

```
def sum_elements(any_list):
    sum = 0

    for x in any_list:
        sum += x

    return sum

my_list = range(1,100,1)
print(sum_elements(my_list))
```

→ 4950

OO program:

```
class ProcessList(object):
    def __init__(self, any_list):
        self.any_list = any_list

    def sum_elements(self):
        self.sum = sum(self.any_list)

my_list = range(1,100,1)
list_sum = ProcessList(my_list)
list_sum.sum_elements()

print(list_sum.sum)
```

Using main() in your Python script

1. Create a file with the name you desire to call your script (e.g. **test.py**)
2. Add your code to test.py:

```
def main():
    print("Hello World!")

if __name__ == "__main__":
    main()
```

3. Save test.py
4. python3 test.py

Using main() in your Python script (cont'd)

```python
from time import sleep

print("This is my file to demonstrate best practices.")

def process_data(data):
    print("Beginning data processing...")
    modified_data = data + " that has been modified"
    sleep(3)
    print("Data processing finished.")
    return modified_data

def read_data_from_web():
    print("Reading data from the Web")
    data = "Data from the web"
    return data

def write_data_to_database(data):
    print("Writing data to a database")
    print(data)

def main():
    data = read_data_from_web()
    modified_data = process_data(data)
    write_data_to_database(modified_data)

if __name__ == "__main__":
    main()
```

Shell

```
$ python3 best_practices.py
This is my file to demonstrate best practices.
Reading data from the Web
Beginning data processing...
Data processing finished.
Writing processed data to a database
Data from the web that has been modified
```

In conclusion

- Python is a very popular versatile programming language that can be used to program following multiple programming paradigms
  - Multi-paradigm language
- It is an interpreted strongly and dynamically typed language
  - JIT compilers for Python exist to improve runtime performance