AASD 4000
Machine Learning - I

Applied AI Solutions Developer Program

Lecture 01
Python Tutorial

Vejey Gandyer

Agenda

Operators

Data Structures

Conditional & Loop Statements

Functions

List Comprehensions

Iterators & Generators

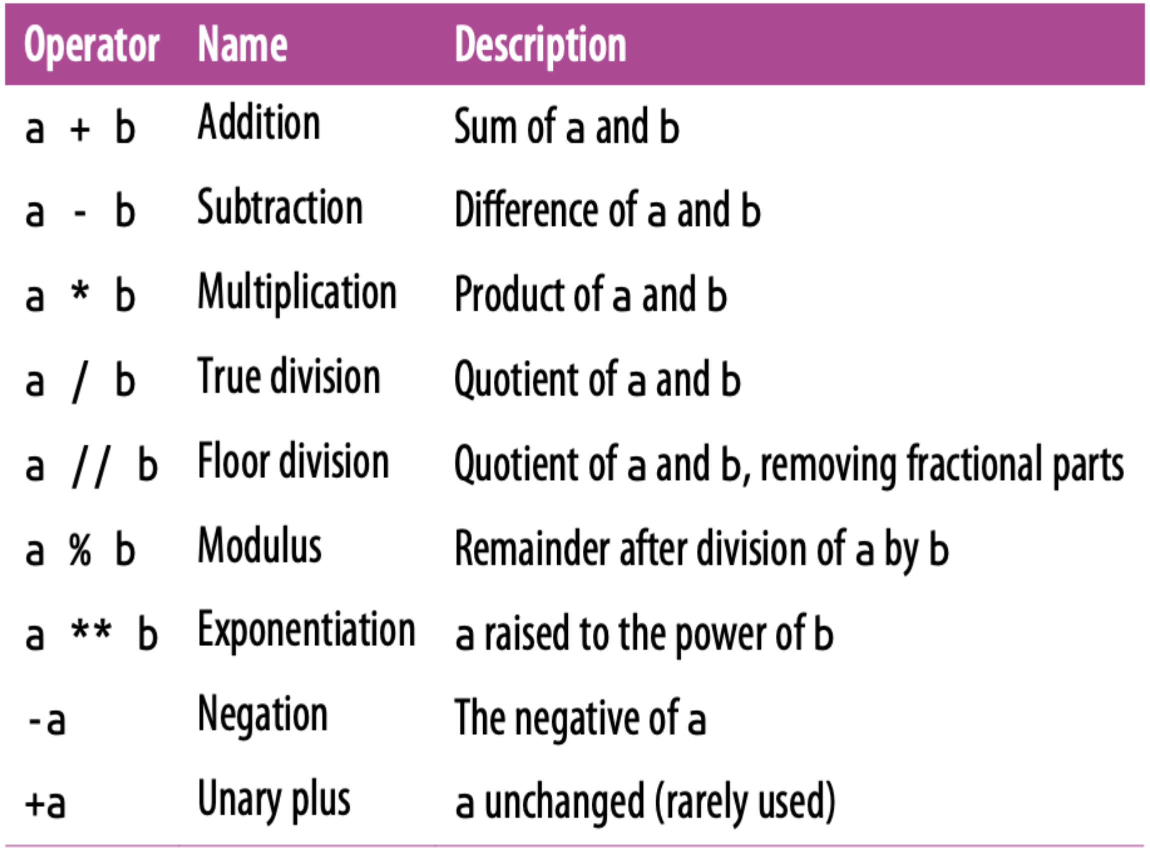
Classes & Objects

Modules & Packages

Files

Operators

Arithmetic Operators



Operator	Name	Description
a + b	Addition	Sum of a and b
a - b	Subtraction	Difference of a and b
a * b	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
a // b	Floor division	Quotient of a and b, removing fractional parts
a % b	Modulus	Remainder after division of a by b
a ** b	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)

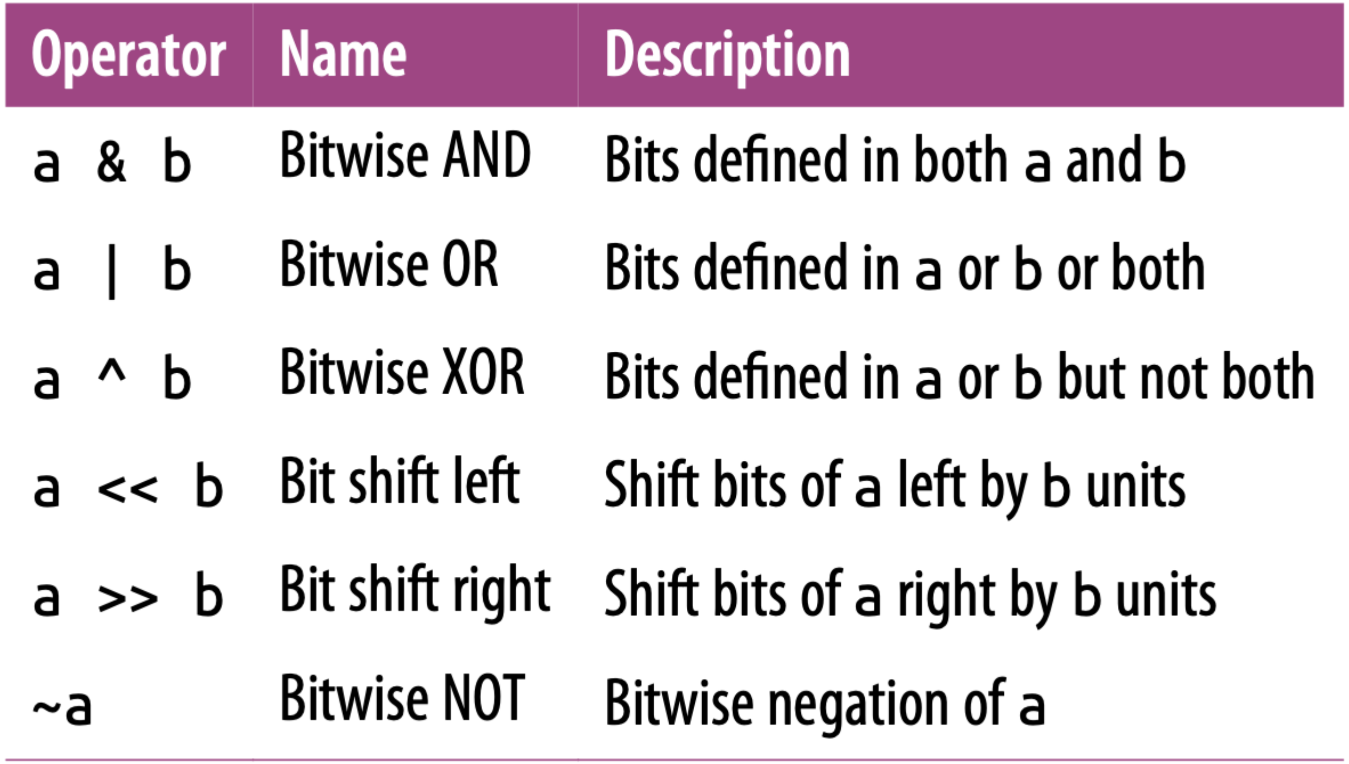
7 Binary Arithmetic Operators

2 Unary Arithmetic Operators

1 New Matrix Operator @

a @ b

Bitwise Operators

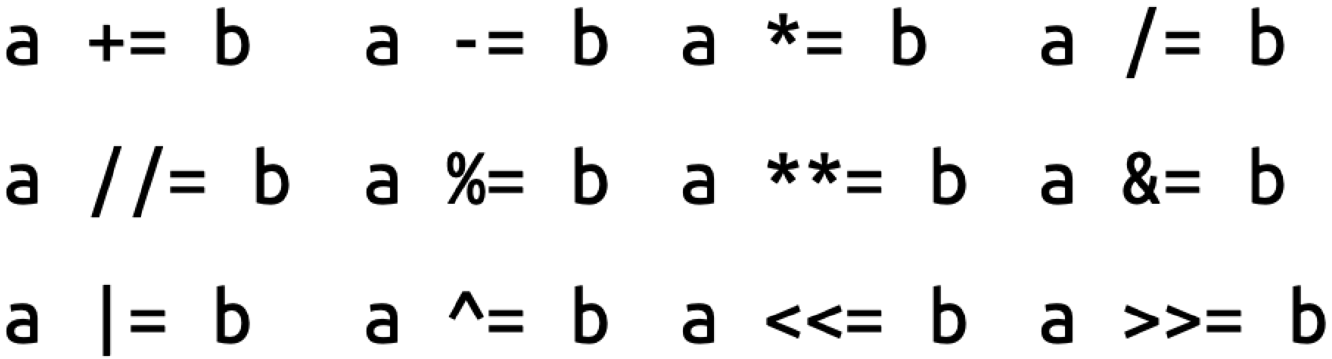


Operator	Name	Description
a & b	Bitwise AND	Bits defined in both a and b
a | b	Bitwise OR	Bits defined in a or b or both
a ^ b	Bitwise XOR	Bits defined in a or b but not both
a << b	Bit shift left	Shift bits of a left by b units
a >> b	Bit shift right	Shift bits of a right by b units
~a	Bitwise NOT	Bitwise negation of a

6 Bitwise Operators

XOR ^ is confused with Exponentiation **

Assignment Operators

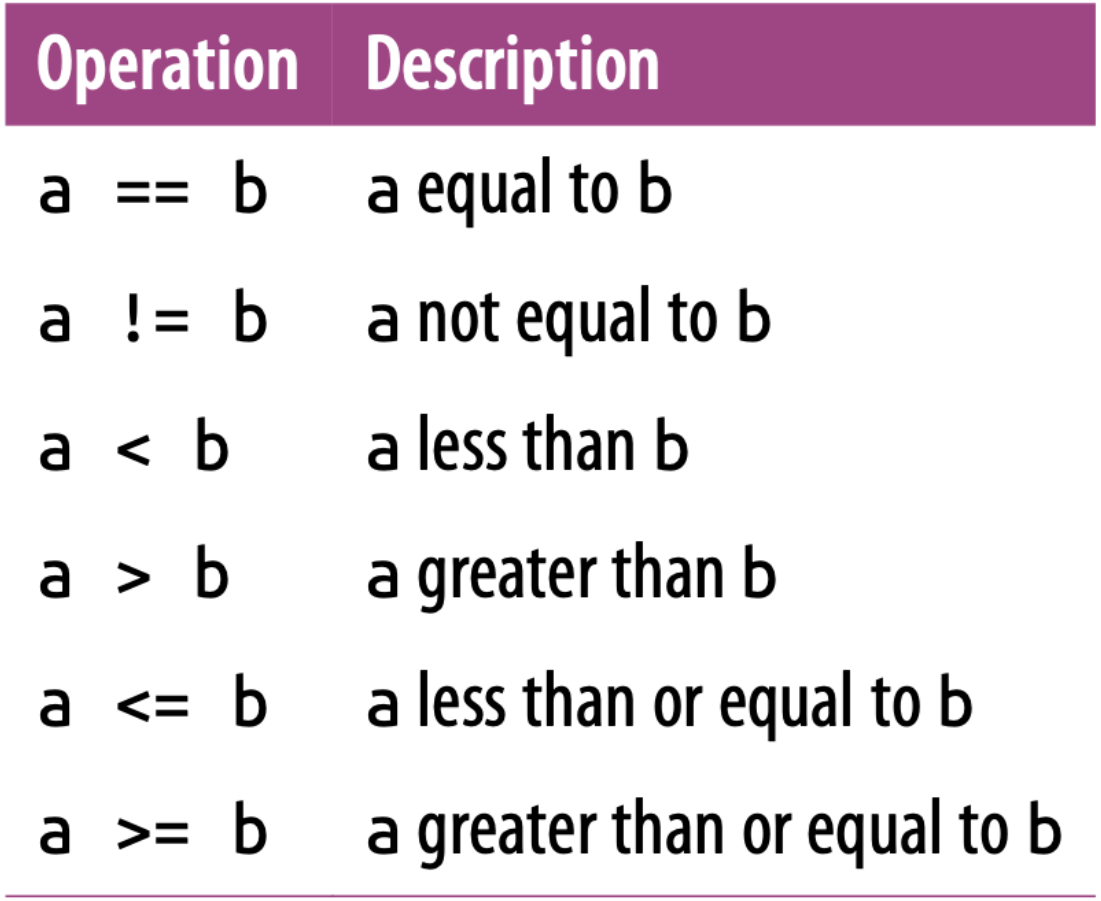


```
a  +=  b     a  -=  b   a  *=  b     a  /=  b
a  //=  b    a  %=  b   a  **=  b    a  &=  b
a  |=  b     a  ^=  b   a  <<=  b    a  >>=  b
```

= operator

Single update and assignment

Comparison Operators

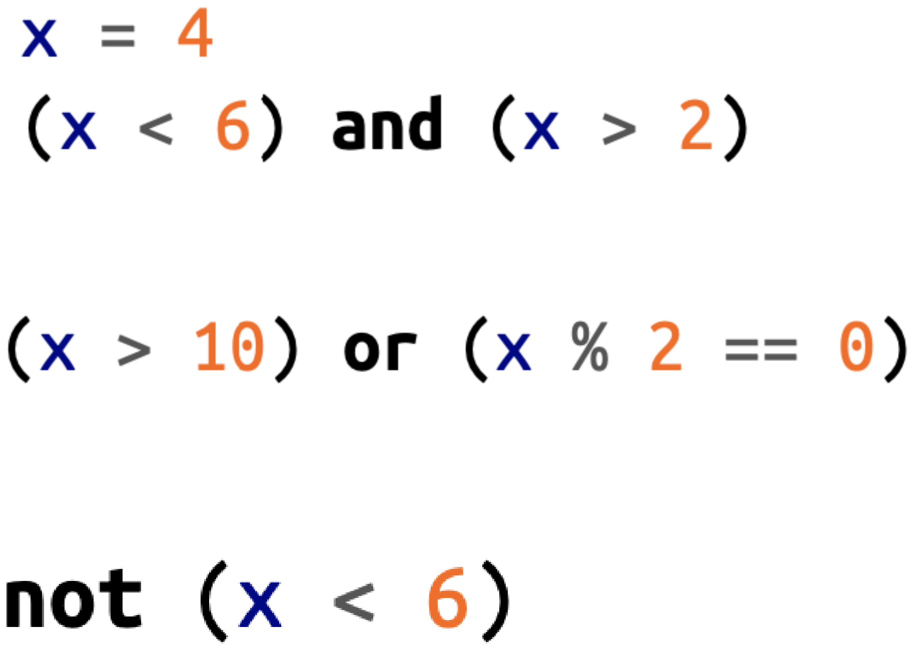


Operation	Description
a == b	a equal to b
a != b	a not equal to b
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

Returns Boolean values

True or False

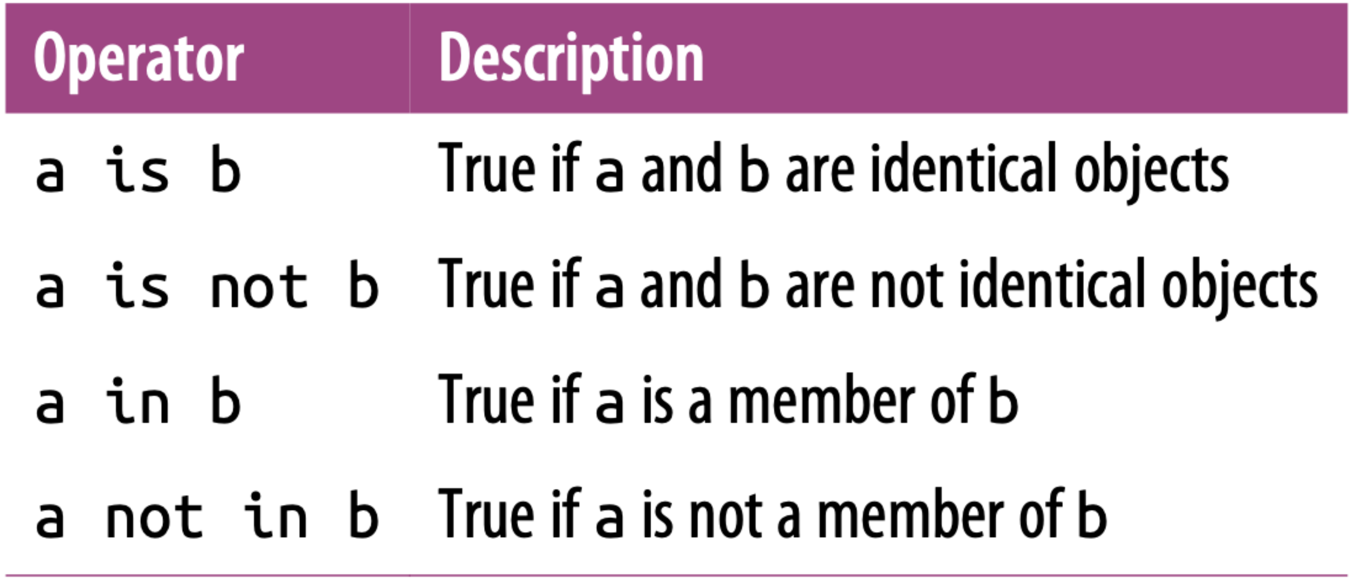
Boolean Operators



```
x = 4
(x < 6) and (x > 2)


(x > 10) or (x % 2 == 0)


not (x < 6)
```

Confusion of when to use Boolean and Bitwise Operators

Identity and Membership Operators



Operator	Description
`a is b`	True if a and b are identical objects
`a is not b`	True if a and b are not identical objects
`a in b`	True if a is a member of b
`a not in b`	True if a is not a member of b

Identity Operator: Checks object identity

Equality is different

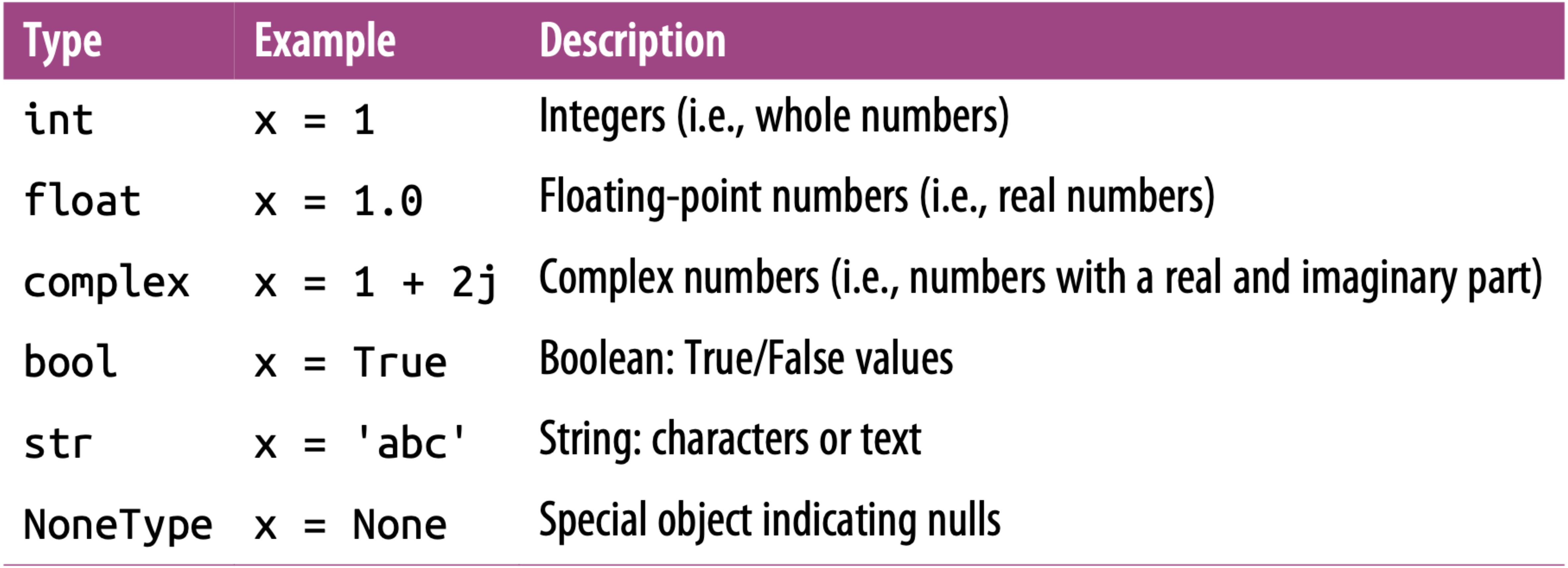
Checks whether two objects (container) are same and not their values

Membership Operator: Checks for membership

Data Structures

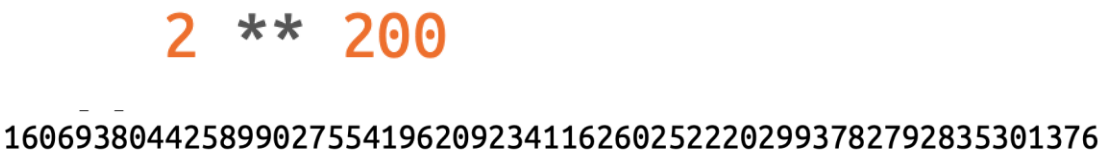
Simple Types

Simple Types



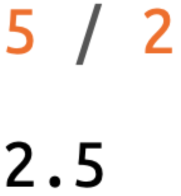
Type	Example	Description
`int`	`x = 1`	Integers (i.e., whole numbers)
`float`	`x = 1.0`	Floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with a real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

Integers



```
2 ** 200
```

```
1606938044258990275541962092341162602522202993782792835301376
```



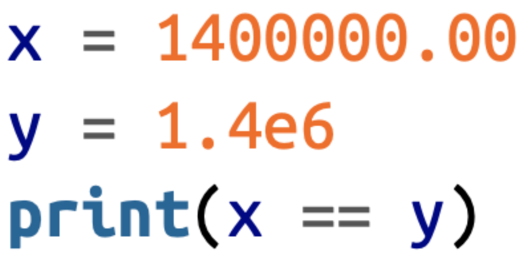
```
5 / 2
```

```
2.5
```

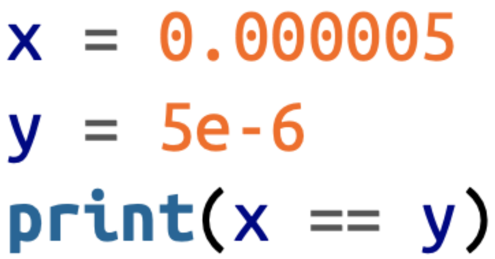
Variable-precision : Better Overflow Management than C, C++

Division Upcasting

Floats



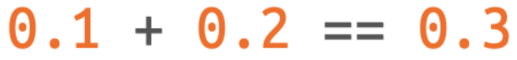
```
x = 1400000.00
y = 1.4e6
print(x == y)
```



```
x = 0.000005
y = 5e-6
print(x == y)
```

Defined both in decimal and exponential notation

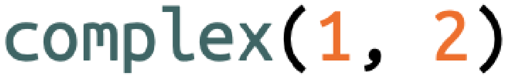
Floating-point precision: rounding-off errors



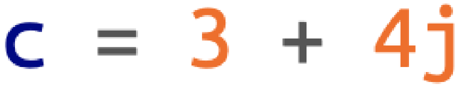
```
0.1 + 0.2 == 0.3
```

Equality Tests

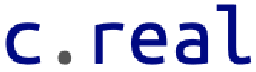
Complex



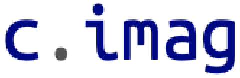
```
complex(1, 2)
```



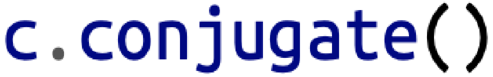
```
c = 3 + 4j
```



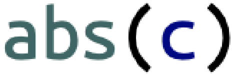
```
c.real
```



```
c.imag
```



```
c.conjugate()
```

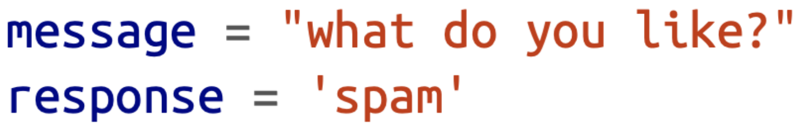


```
abs(c)
```

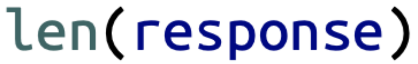
Contains both Real and Imaginary parts

Methods operating on Complex numbers

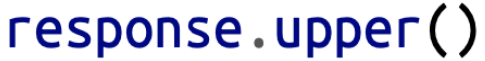
String Type



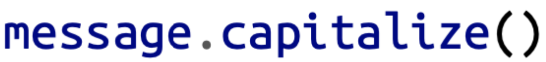
```python
message = "what do you like?"
response = 'spam'
```



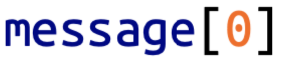
```python
len(response)
```



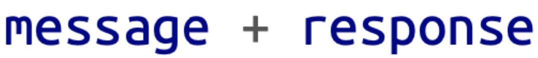
```python
response.upper()
```



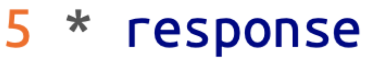
```python
message.capitalize()
```



```python
message[0]
```



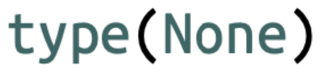
```python
message + response
```



```python
5 * response
```

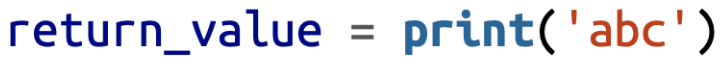
Several string manipulations are possible

None Type



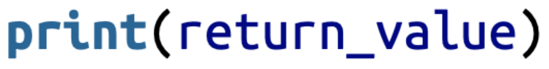
```
type(None)
```

NoneType has only one value: None



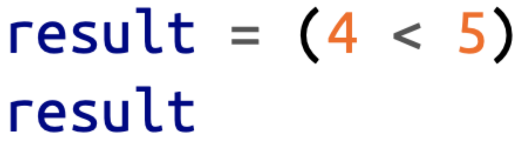
```
return_value = print('abc')
```

Default return value of a function eg: print()

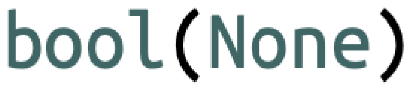


```
print(return_value)
```

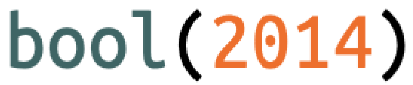
Boolean Type



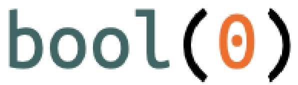
```python
result = (4 < 5)
result
```



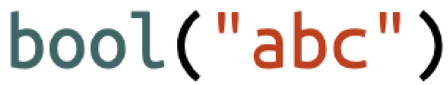
```python
bool(None)
```



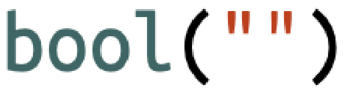
```python
bool(2014)
```



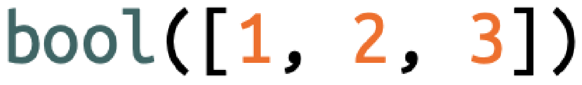
```python
bool(0)
```



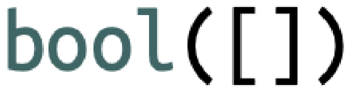
```python
bool("abc")
```



```python
bool("")
```



```python
bool([1, 2, 3])
```



```python
bool([])
```

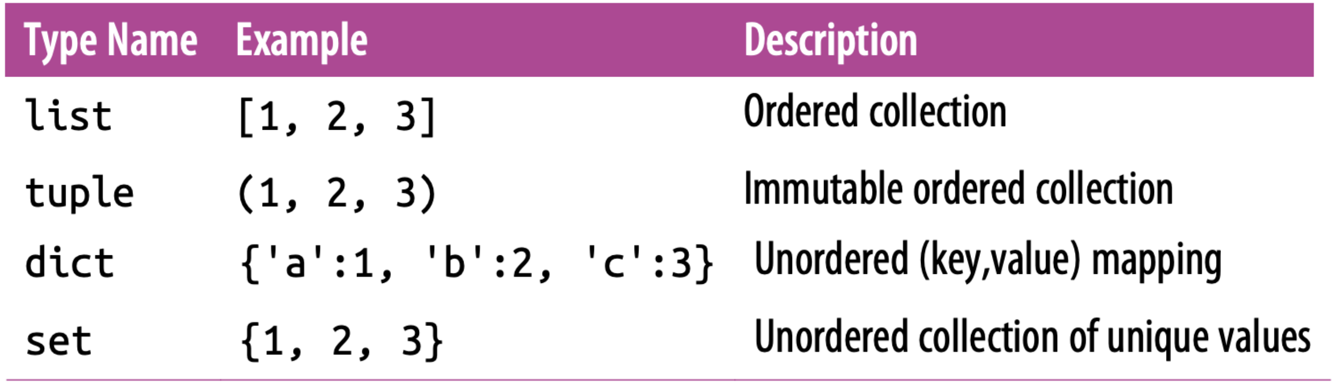
Boolean Type has only two values: True & False

bool() constructor

Data Structures

Compound Types

Compound Types



Type Name	Example	Description
list	[1, 2, 3]	Ordered collection
tuple	(1, 2, 3)	Immutable ordered collection
dict	{'a':1, 'b':2, 'c':3}	Unordered (key,value) mapping
set	{1, 2, 3}	Unordered collection of unique values

4 Compound Types

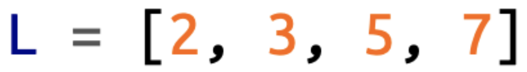
List - [ ]

Tuple - ( )

Dict - { }

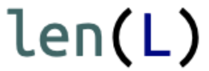
Set - { }

Lists

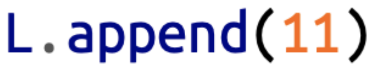


```
L = [2, 3, 5, 7]
```

[ ]

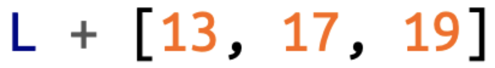


```
len(L)
```

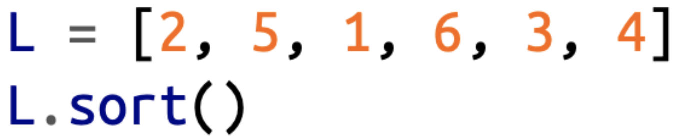


```
L.append(11)
```

Mutable data structure

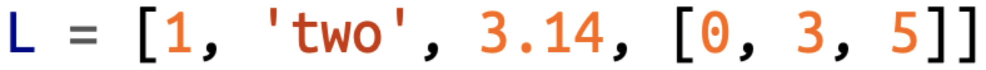


```
L + [13, 17, 19]
```



```
L = [2, 5, 1, 6, 3, 4]
L.sort()
```

Ordered data structure



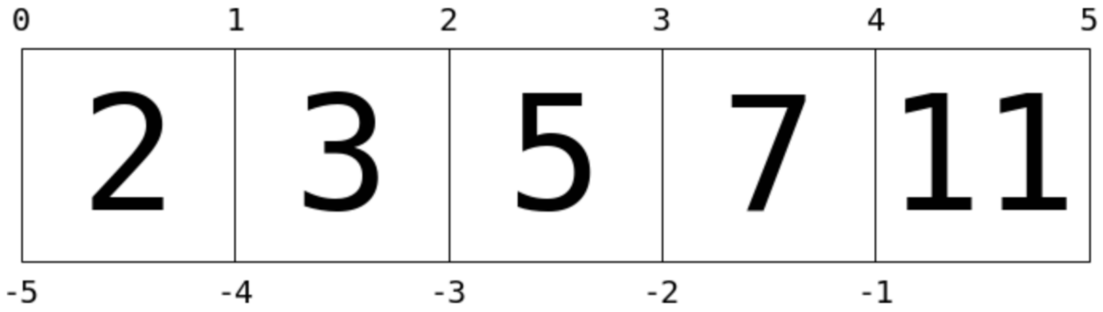
```
L = [1, 'two', 3.14, [0, 3, 5]]
```

Multi-type data structure

Lists: Indexing and Slicing

```
L = [2, 3, 5, 7, 11]
```

```
L[0]      L[1]      L[-1]      L[-2]
```



```
L[0:3]        L[:3]        L[-3:]
```

```
L[::2]                L[::-1]
```

Indexing:

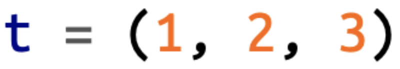
Zero-based

Accesses single values

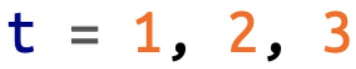
Slicing:

Accesses multiple values

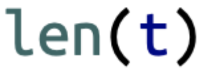
Tuples



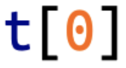
```
t = (1, 2, 3)
```



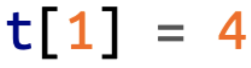
```
t = 1, 2, 3
```



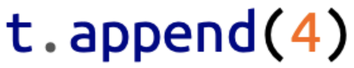
```
len(t)
```



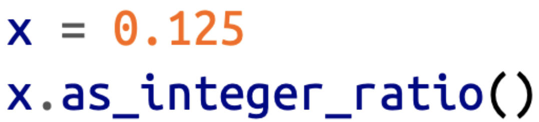
```
t[0]
```



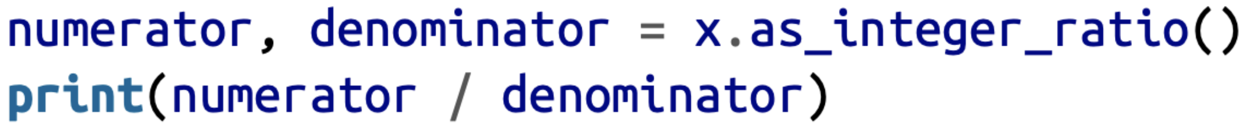
```
t[1] = 4
```



```
t.append(4)
```



```
x = 0.125
x.as_integer_ratio()
```



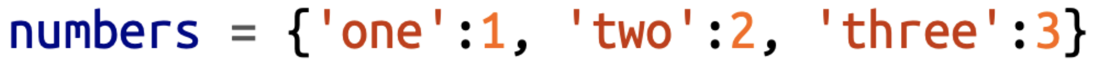
```
numerator, denominator = x.as_integer_ratio()
print(numerator / denominator)
```

( )

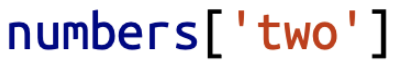
Immutable Data Structure

Used in Functions returning multiple return values

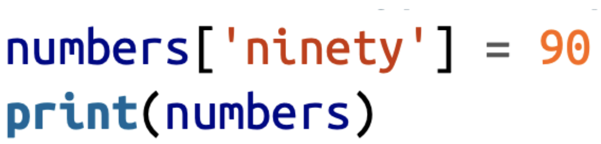
Dictionaries



```
numbers = {'one':1, 'two':2, 'three':3}
```



```
numbers['two']
```



```
numbers['ninety'] = 90
print(numbers)
```

{ }

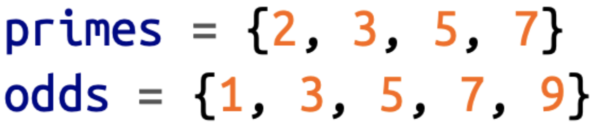
Mutable Data Structure

Mappings of Keys and Values

Unordered Data Structure

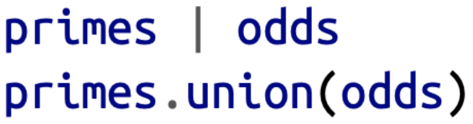
Efficient Data Structure

Sets



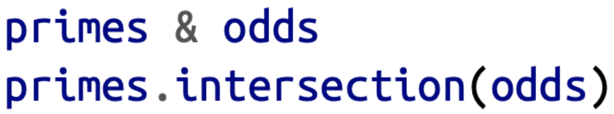
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
```

{ }



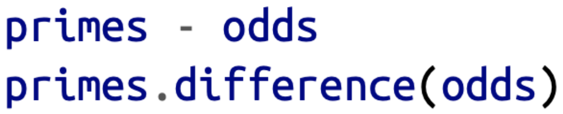
```
primes | odds
primes.union(odds)
```

Unordered collection of unique items

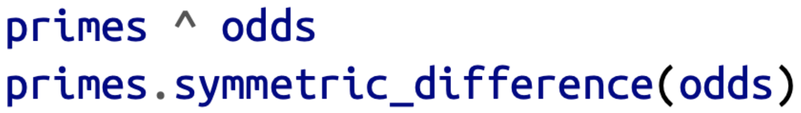


```
primes & odds
primes.intersection(odds)
```

Operations: Union, Intersection, Difference, Symmetric difference

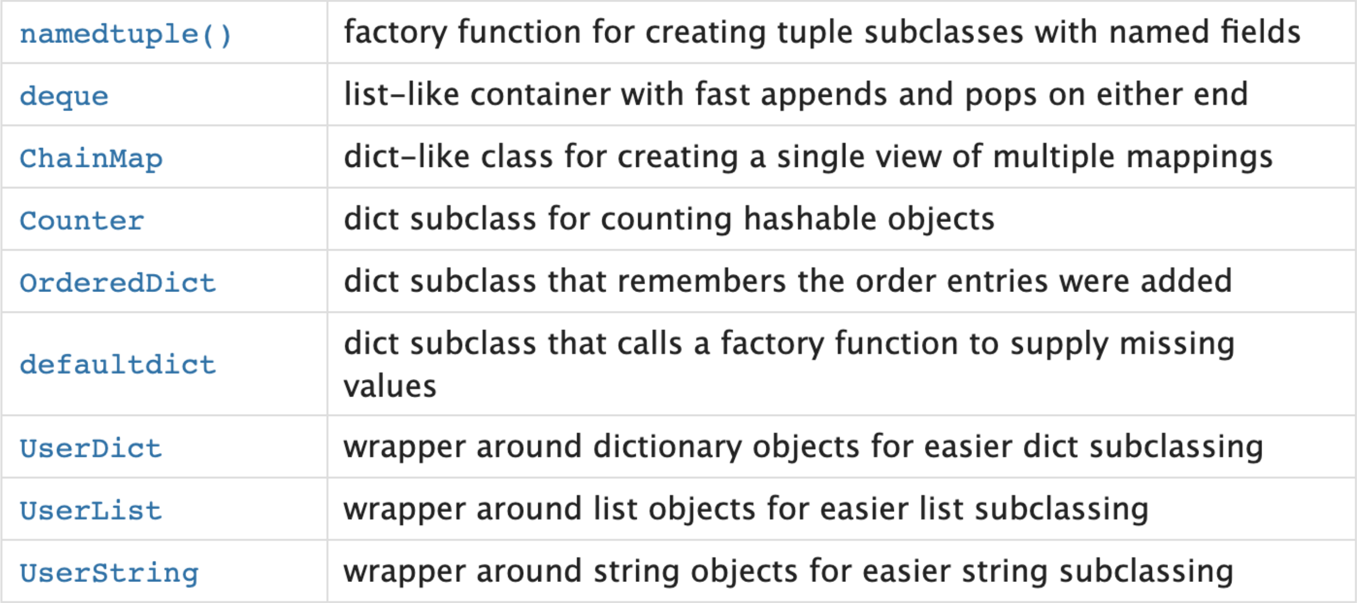


```
primes - odds
primes.difference(odds)
```



```
primes ^ odds
primes.symmetric_difference(odds)
```

Collections



`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

Powerful built-in collections

deque

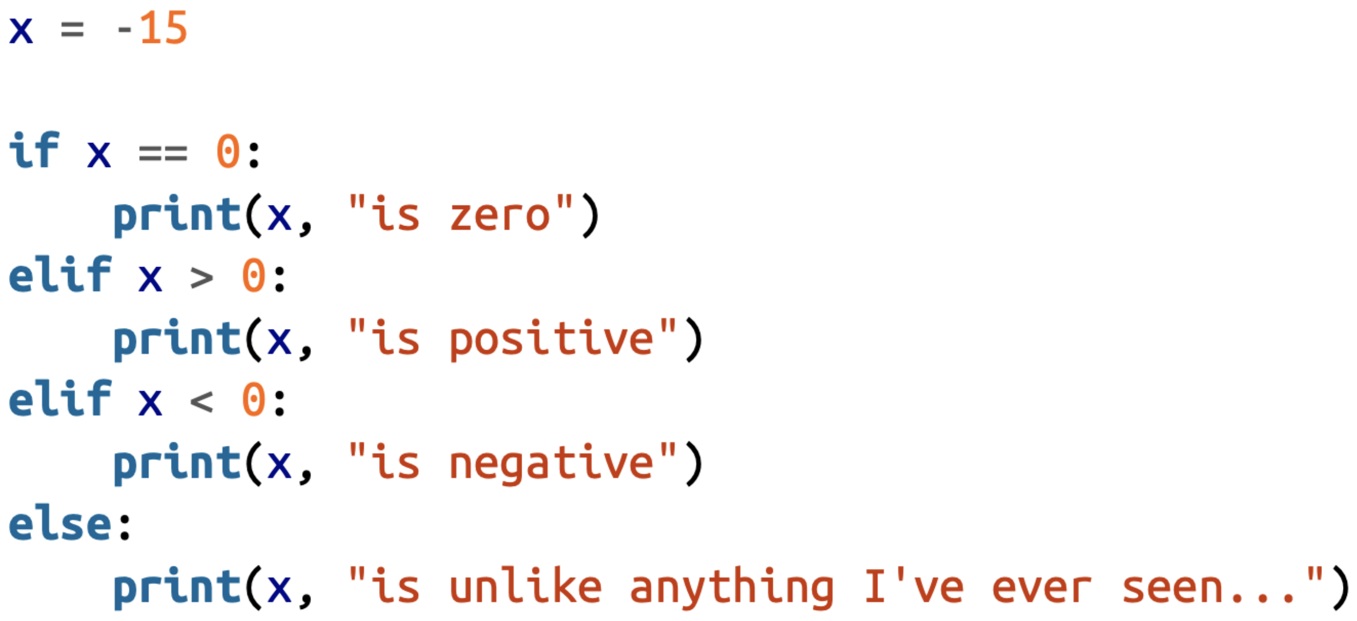
OrderedDict

Counter

https://docs.python.org/3/library/collections.html

Conditional & Loop Statements

Conditional Statements



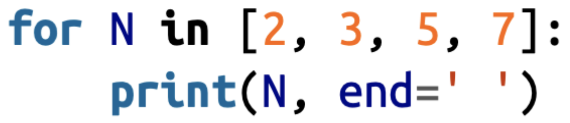
```python
x = -15

if x == 0:
    print(x, "is zero")
elif x > 0:
    print(x, "is positive")
elif x < 0:
    print(x, "is negative")
else:
    print(x, "is unlike anything I've ever seen...")
```

If-then statements

Unique elif

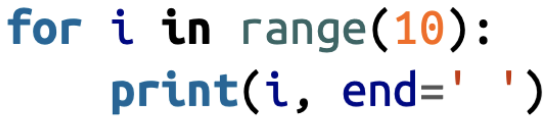
Executes a piece of code when a condition is met

for loop



```python
for N in [2, 3, 5, 7]:
    print(N, end=' ')
```

Executes some piece of code repeatedly



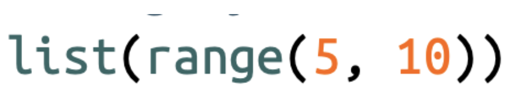
```python
for i in range(10):
    print(i, end=' ')
```

Variable to use

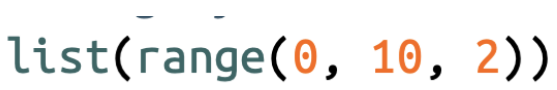
Sequence to loop over

in operator

Iterable object (range( ))

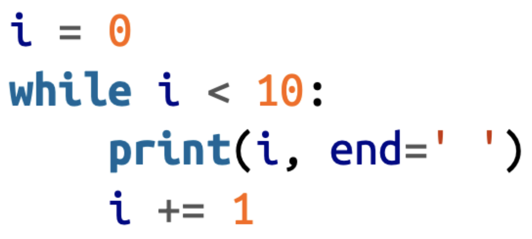


```python
list(range(5, 10))
```



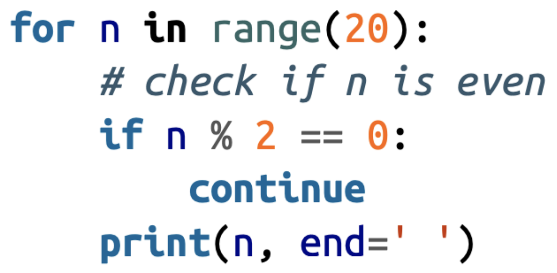
```python
list(range(0, 10, 2))
```

while loop

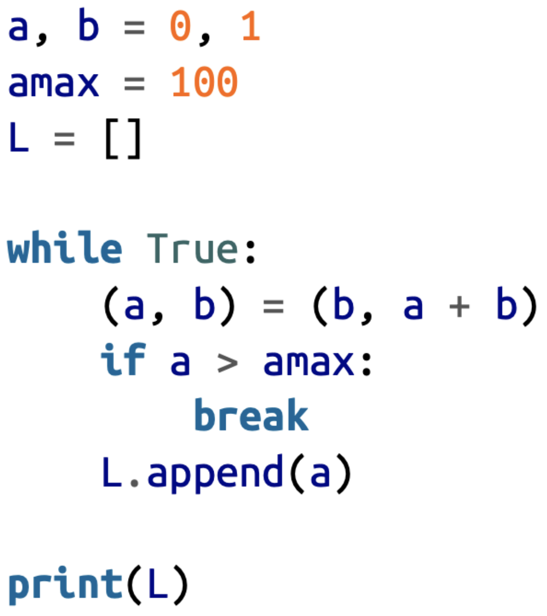


```python
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

Executes some piece of code until condition is met



```python
for n in range(20):
    # check if n is even
    if n % 2 == 0:
        continue
    print(n, end=' ')
```



```python
a, b = 0, 1
amax = 100
L = []

while True:
    (a, b) = (b, a + b)
    if a > amax:
        break
    L.append(a)

print(L)
```

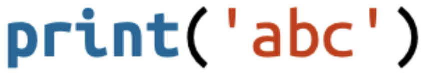
break: breaks out of the loop

continue: skips the remainder of current iteration and goes to next iteration

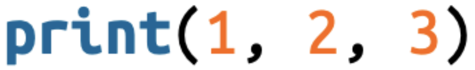
pass: goes to the next statement

Functions

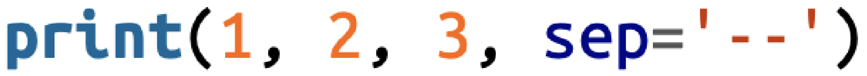
Functions



```
print('abc')
```



```
print(1, 2, 3)
```



```
print(1, 2, 3, sep='--')
```

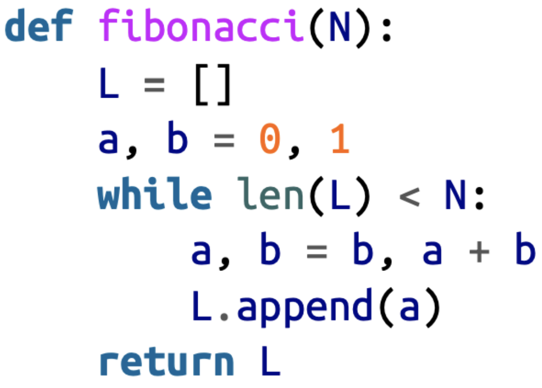
def

Function name

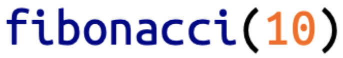
Function arguments

Keyword arguments

Defining Functions

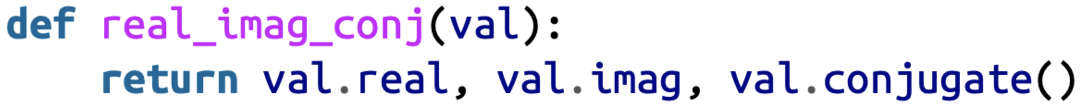


```python
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

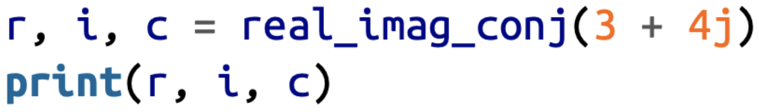


```python
fibonacci(10)
```

Logic can be encapsulated within a reusable piece of code called function



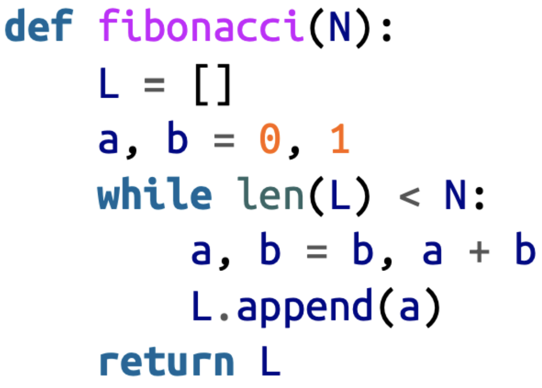
```python
def real_imag_conj(val):
    return val.real, val.imag, val.conjugate()
```



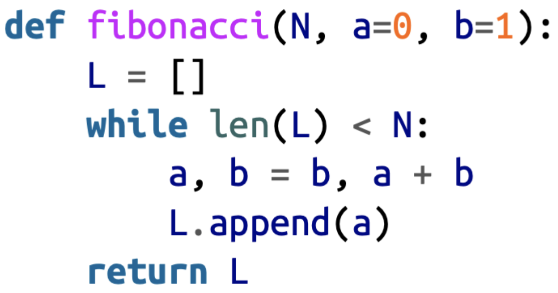
```python
r, i, c = real_imag_conj(3 + 4j)
print(r, i, c)
```

Single or multiple return values are possible

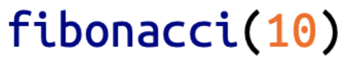
Default Arguments



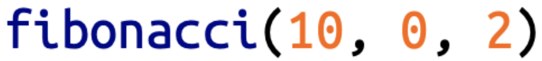
```python
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```



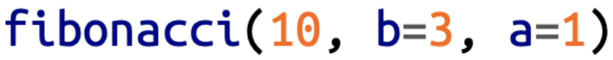
```python
def fibonacci(N, a=0, b=1):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```



```python
fibonacci(10)
```



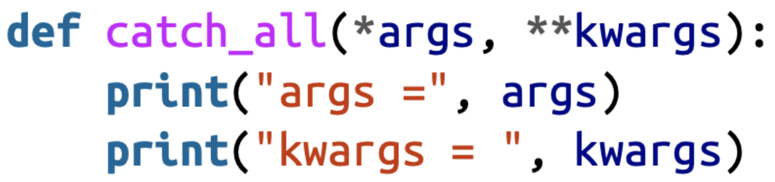
```python
fibonacci(10, 0, 2)
```



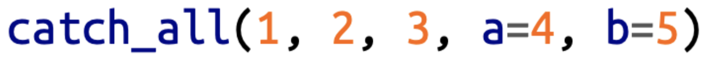
```python
fibonacci(10, b=3, a=1)
```

Certain values should be used most of the times inside the logic of a function, but needs to give the user the flexibility

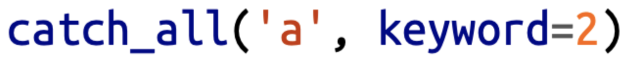
*args & **kwargs



```python
def catch_all(*args, **kwargs):
    print("args =", args)
    print("kwargs = ", kwargs)
```



```python
catch_all(1, 2, 3, a=4, b=5)
```



```python
catch_all('a', keyword=2)
```

Function with unknown number of arguments

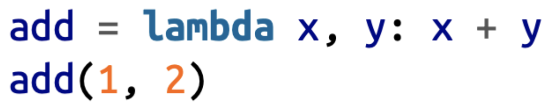
*args: Arguments

Expand this as a Sequence

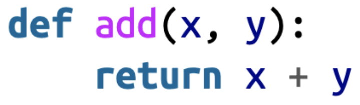
**kwargs: Keyword Arguments

Expand this as a Dictionary

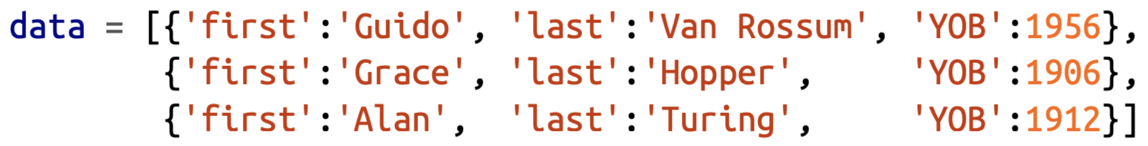
Lambda function



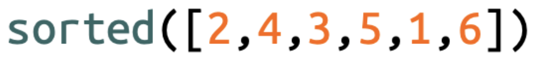
```python
add = lambda x, y: x + y
add(1, 2)
```



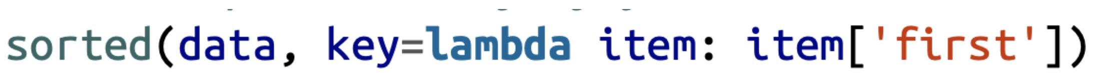
```python
def add(x, y):
    return x + y
```



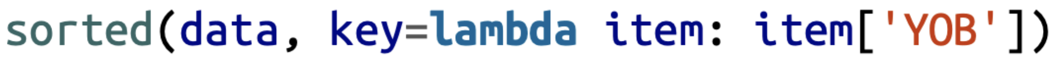
```python
data = [{'first':'Guido', 'last':'Van Rossum', 'YOB':1956},
        {'first':'Grace', 'last':'Hopper',     'YOB':1906},
        {'first':'Alan',  'last':'Turing',     'YOB':1912}]
```



```python
sorted([2,4,3,5,1,6])
```



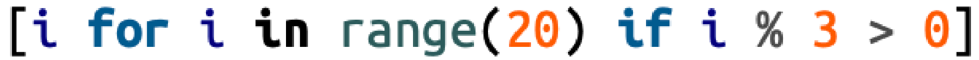
```python
sorted(data, key=lambda item: item['first'])
```



```python
sorted(data, key=lambda item: item['YOB'])
```

List Comprehensions

List Comprehensions

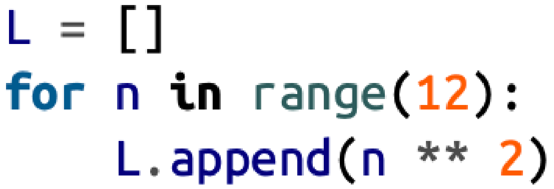


```
[i for i in range(20) if i % 3 > 0]
```

Compress a large set of for loop logic into one line

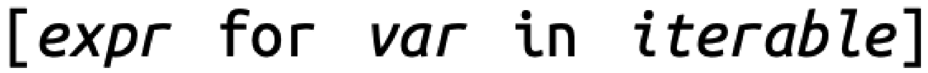
Pythonic way of writing a program

List Comprehensions

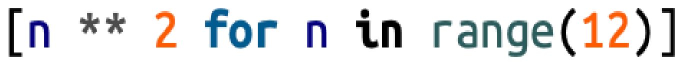


```python
L = []
for n in range(12):
    L.append(n ** 2)
```

Construct a list comprehension for this logic

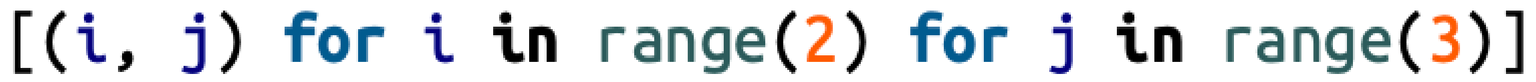


```
[expr for var in iterable]
```

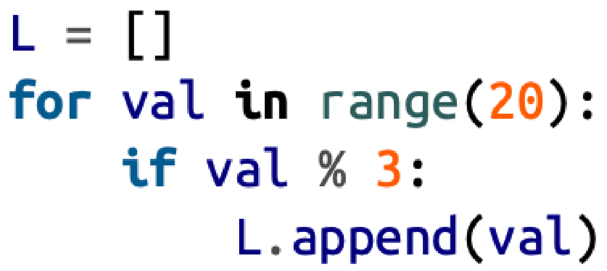


```python
[n ** 2 for n in range(12)]
```

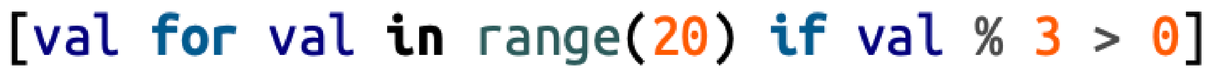
List Comprehensions



```
[(i, j) for i in range(2) for j in range(3)]
```

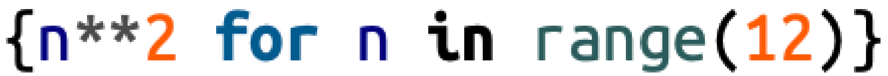


```
L = []
for val in range(20):
    if val % 3:
        L.append(val)
```

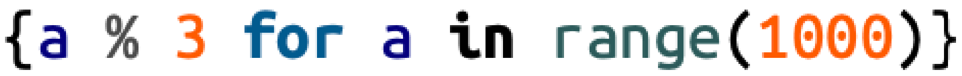


```
[val for val in range(20) if val % 3 > 0]
```

Set Comprehensions

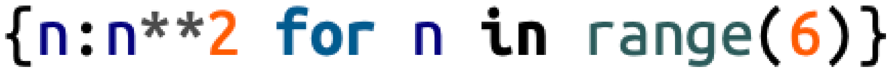


```
{n**2 for n in range(12)}
```



```
{a % 3 for a in range(1000)}
```

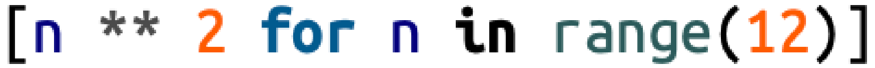
Dict Comprehensions



```
{n:n**2 for n in range(6)}
```

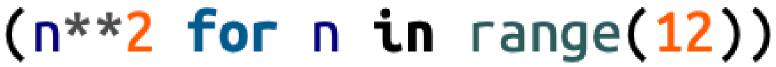
Generators

List Comprehensions



```
[n ** 2 for n in range(12)]
```

Generator Expressions



```
(n**2 for n in range(12))
```

Generator expression is a list comprehension in which elements are generated as needed rather than all at once

List Comprehensions Vs Generators

List comprehension is a collection of values

Memory is allocated when creating a list

Size of a list is limited

Can be iterated multiple times

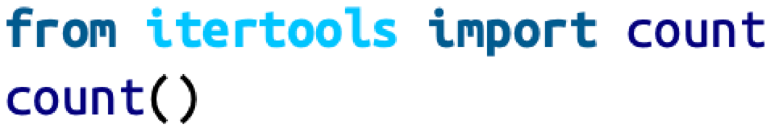
Generator expression is a recipe for producing values

Memory is not allocated until it is asked for computation

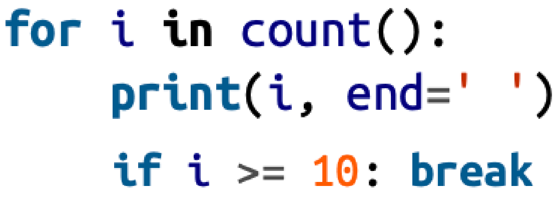
Unlimited size

Only one iteration

Infinite number generator



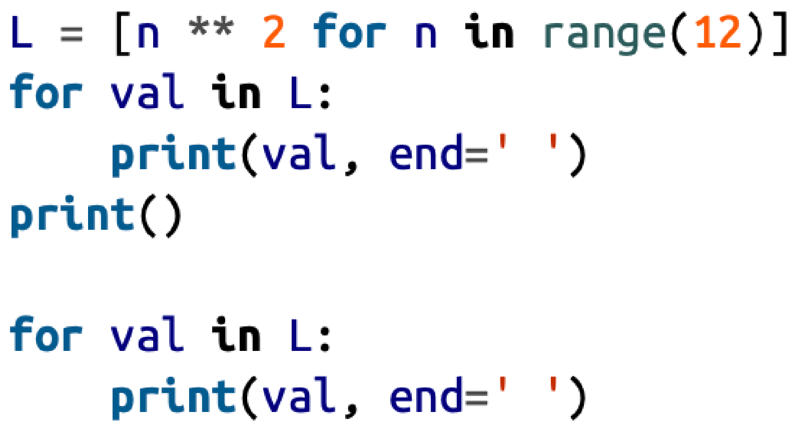
```python
from itertools import count
count()
```



```python
for i in count():
    print(i, end=' ')
    if i >= 10: break
```

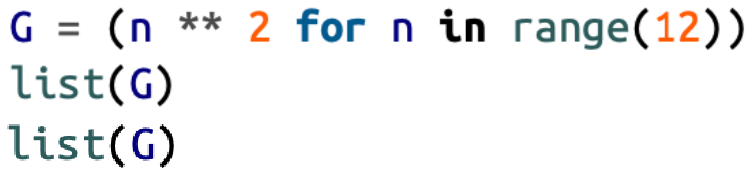
Only one iteration

List Comprehension



```
L = [n ** 2 for n in range(12)]
for val in L:
    print(val, end=' ')
print()

for val in L:
    print(val, end=' ')
```

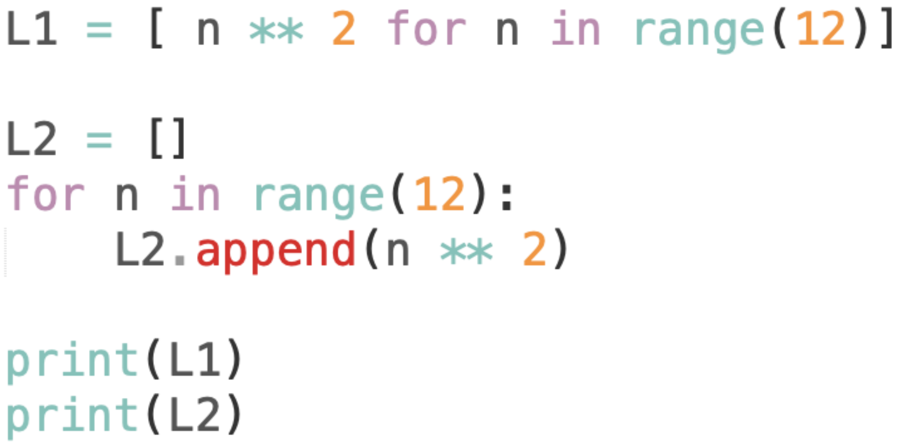
Generator



```
G = (n ** 2 for n in range(12))
list(G)
list(G)
```

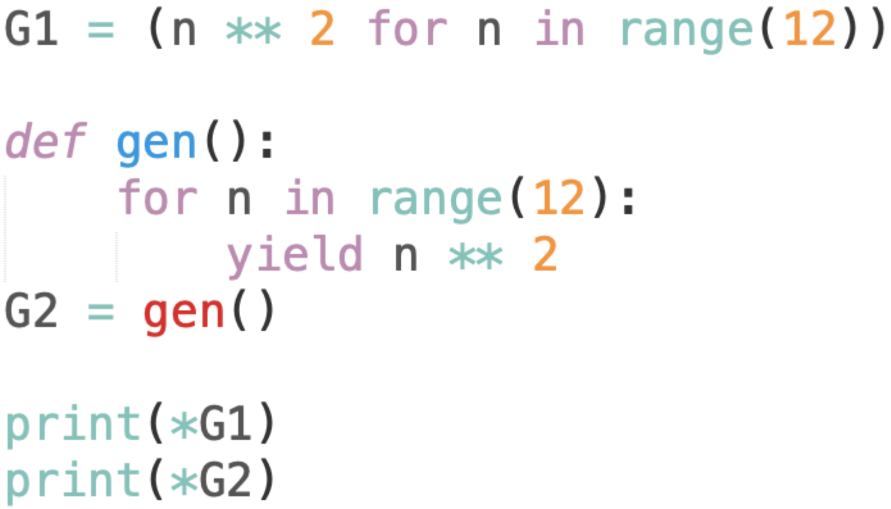
yield - yields a sequence of values

List Comprehension



```
L1 = [ n ** 2 for n in range(12)]

L2 = []
for n in range(12):
    L2.append(n ** 2)

print(L1)
print(L2)
```

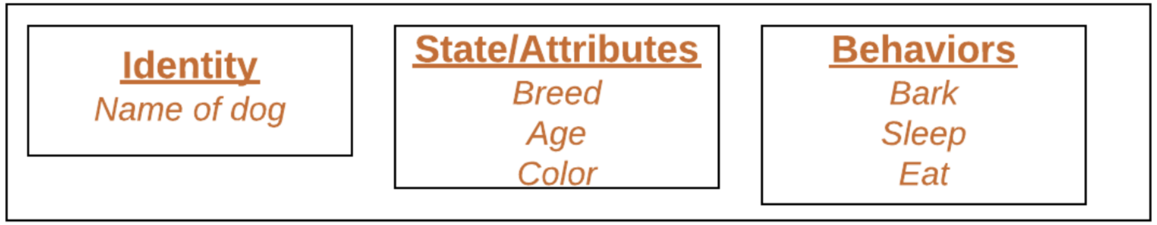
Generator



```
G1 = (n ** 2 for n in range(12))

def gen():
    for n in range(12):
        yield n ** 2
G2 = gen()

print(*G1)
print(*G2)
```

Classes & Objects

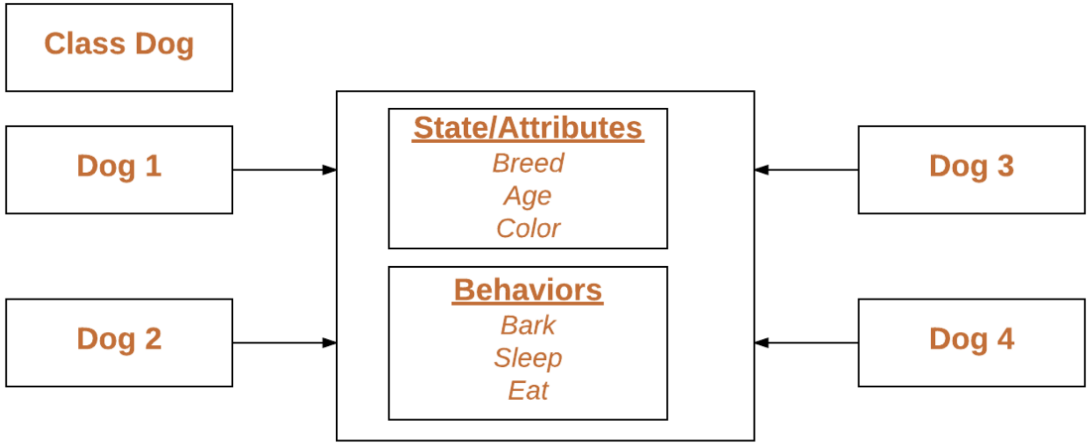
Classes & Objects



Identity
Name of dog

State/Attributes
Breed
Age
Color

Behaviors
Bark
Sleep
Eat



Class Dog

Dog 1

Dog 2

State/Attributes
Breed
Age
Color

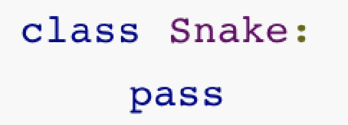
Behaviors
Bark
Sleep
Eat

Dog 3

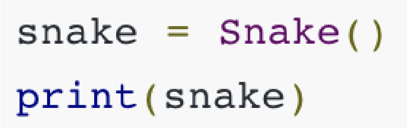
Dog 4

- State : It is represented by attributes of an object. It also reflects the properties of an object.
- Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.
- Identity : It gives a unique name to an object and enables one object to interact with other objects

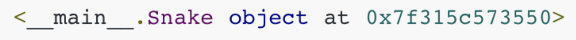
Classes & Objects



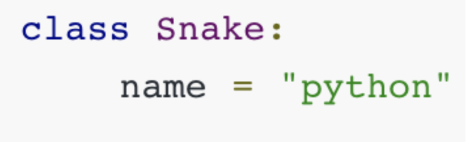
```
class Snake:
    pass
```



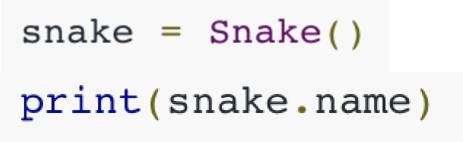
```
snake = Snake()
print(snake)
```



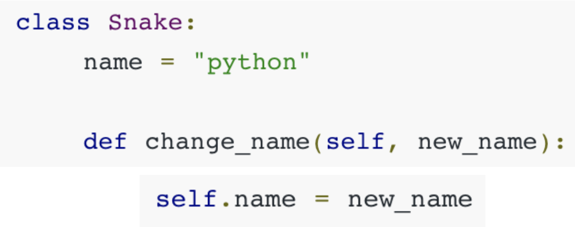
```
<__main__.Snake object at 0x7f315c573550>
```



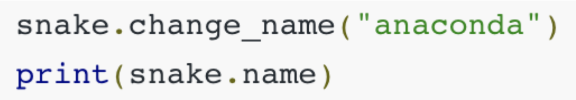
```
class Snake:
    name = "python"
```



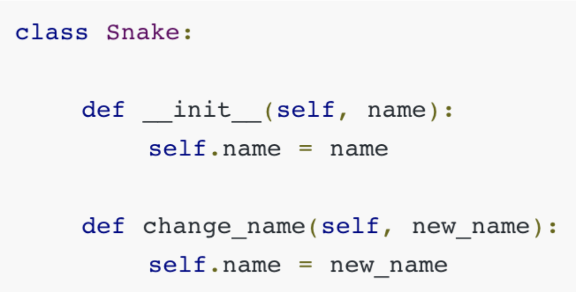
```
snake = Snake()
print(snake.name)
```



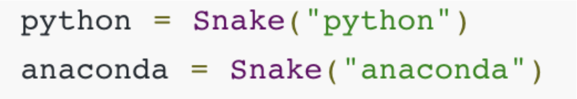
```
class Snake:
    name = "python"

    def change_name(self, new_name):
        self.name = new_name
```



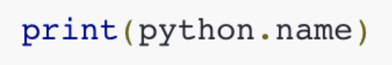
```
snake.change_name("anaconda")
print(snake.name)
```



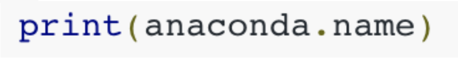
```
class Snake:

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

    def change_name(self, new_name):
        self.name = new_name
```



```
python = Snake("python")
anaconda = Snake("anaconda")
```



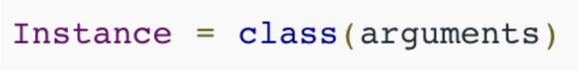
```
print(python.name)
```



```
print(anaconda.name)
```

Class: code template for creating objects, created by keyword **class**

Objects: member variables and have behaviour associated with them, created using **constructor** of the clas

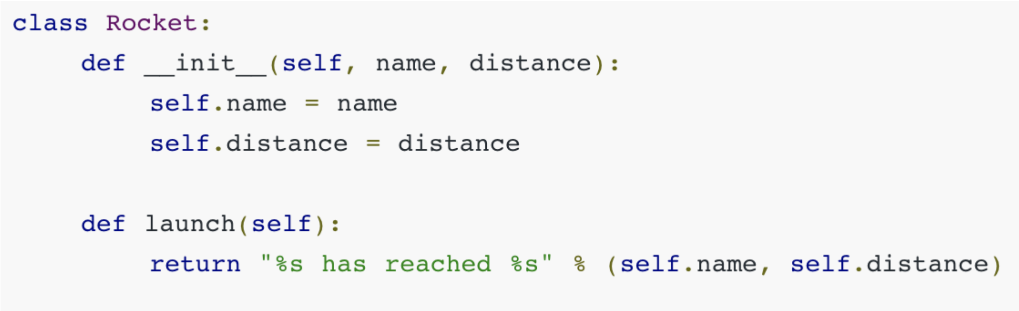


```
Instance = class(arguments)
```

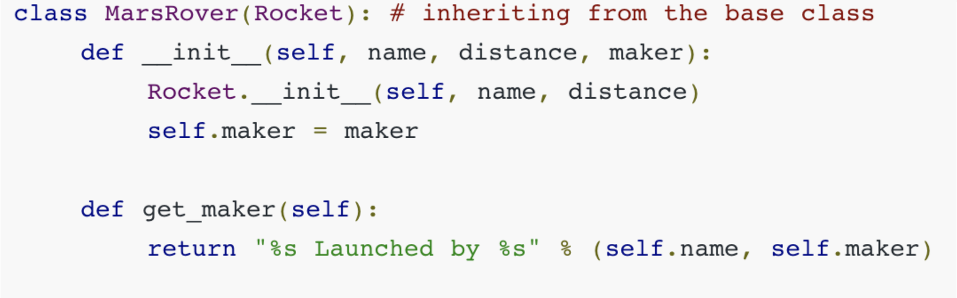
Attributes: Properties inside a class

Methods: Operations on attributes

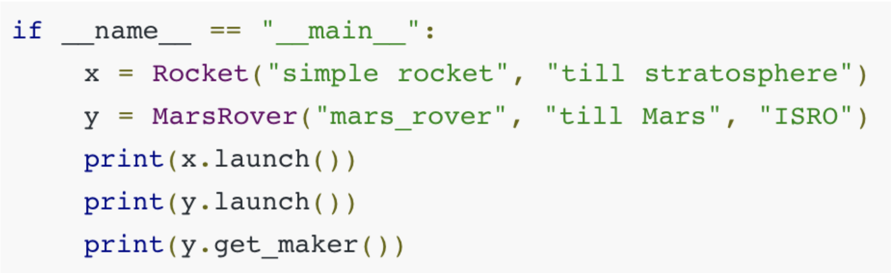
Inheritance



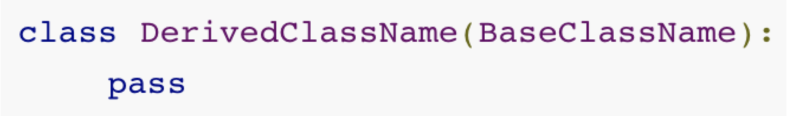
```python
class Rocket:
    def __init__(self, name, distance):
        self.name = name
        self.distance = distance

    def launch(self):
        return "%s has reached %s" % (self.name, self.distance)
```



```python
class MarsRover(Rocket): # inheriting from the base class
    def __init__(self, name, distance, maker):
        Rocket.__init__(self, name, distance)
        self.maker = maker

    def get_maker(self):
        return "%s Launched by %s" % (self.name, self.maker)
```



```python
if __name__ == "__main__":
    x = Rocket("simple rocket", "till stratosphere")
    y = MarsRover("mars_rover", "till Mars", "ISRO")
    print(x.launch())
    print(y.launch())
    print(y.get_maker())
```



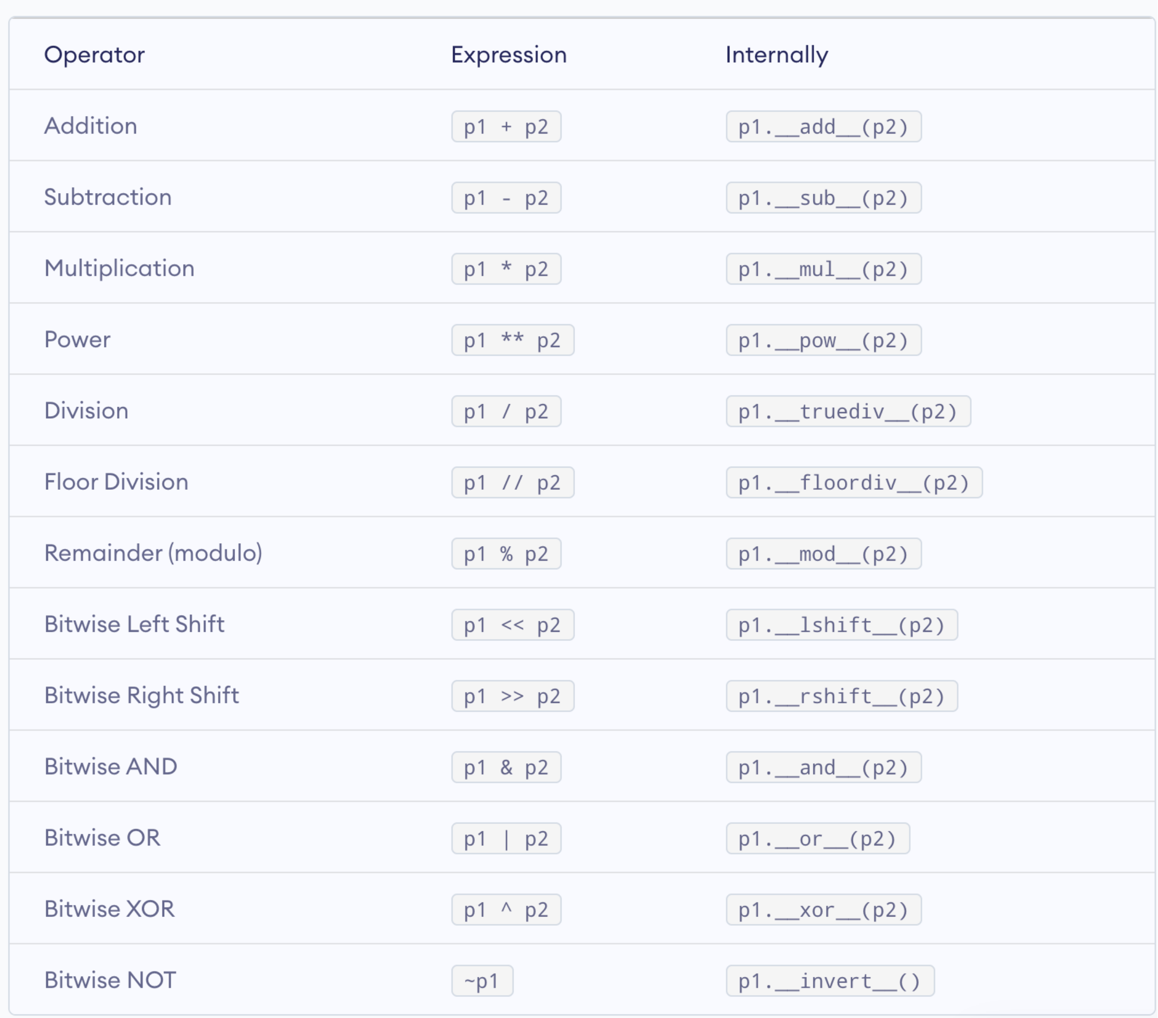
```python
class DerivedClassName(BaseClassName):
    pass
```

Inheritance: an object is based on another object, methods and attributes that were defined in the base class will also be present in the inherited class

Abstract away similar code in multiple classes

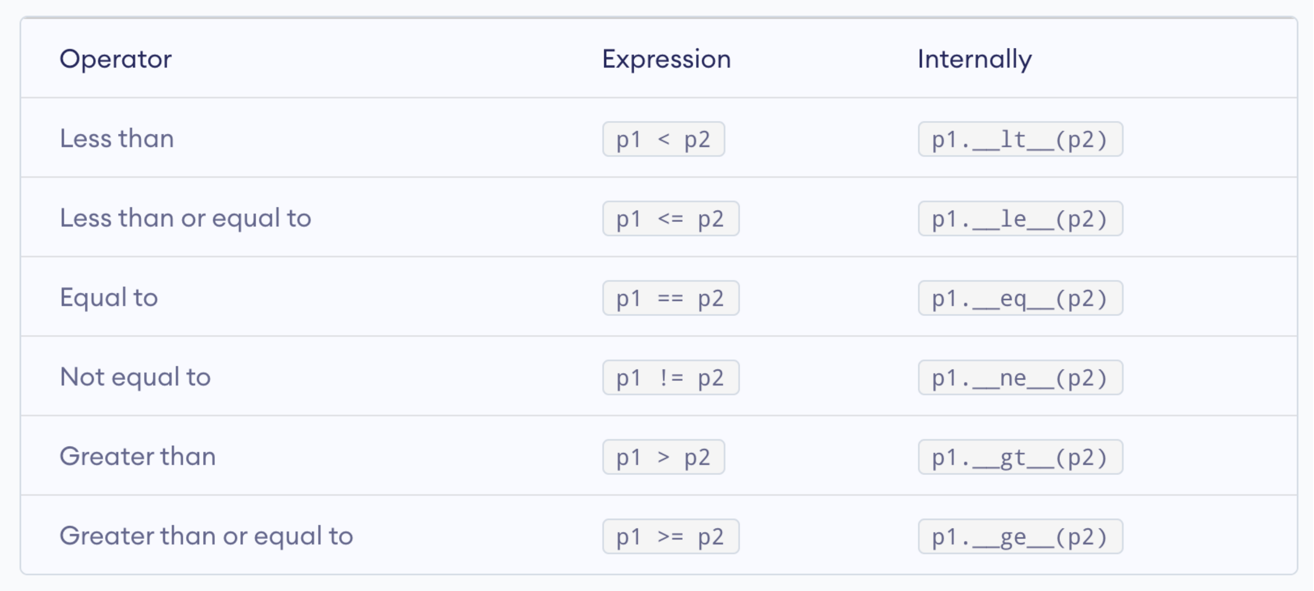
The abstracted code will reside in the base class and the previous classes will now inherit from the base class

Operator Overloading



Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Overloading of operators

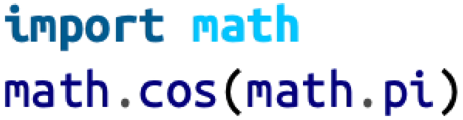


Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Modules & Packages

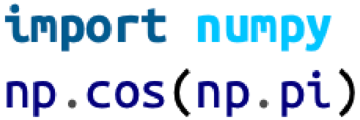
Loading Modules

Explicit module import



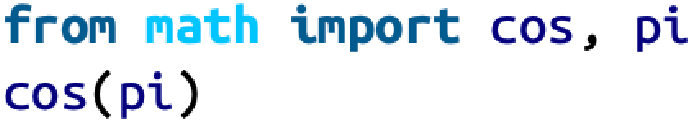
```
import math
math.cos(math.pi)
```

Alias Explicit module import



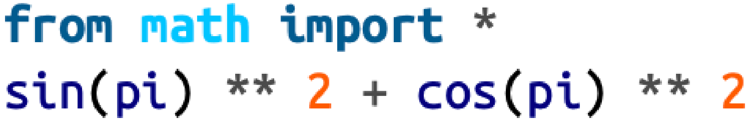
```
import numpy as np
np.cos(np.pi)
```

Explicit import of module contents



```
from math import cos, pi
cos(pi)
```

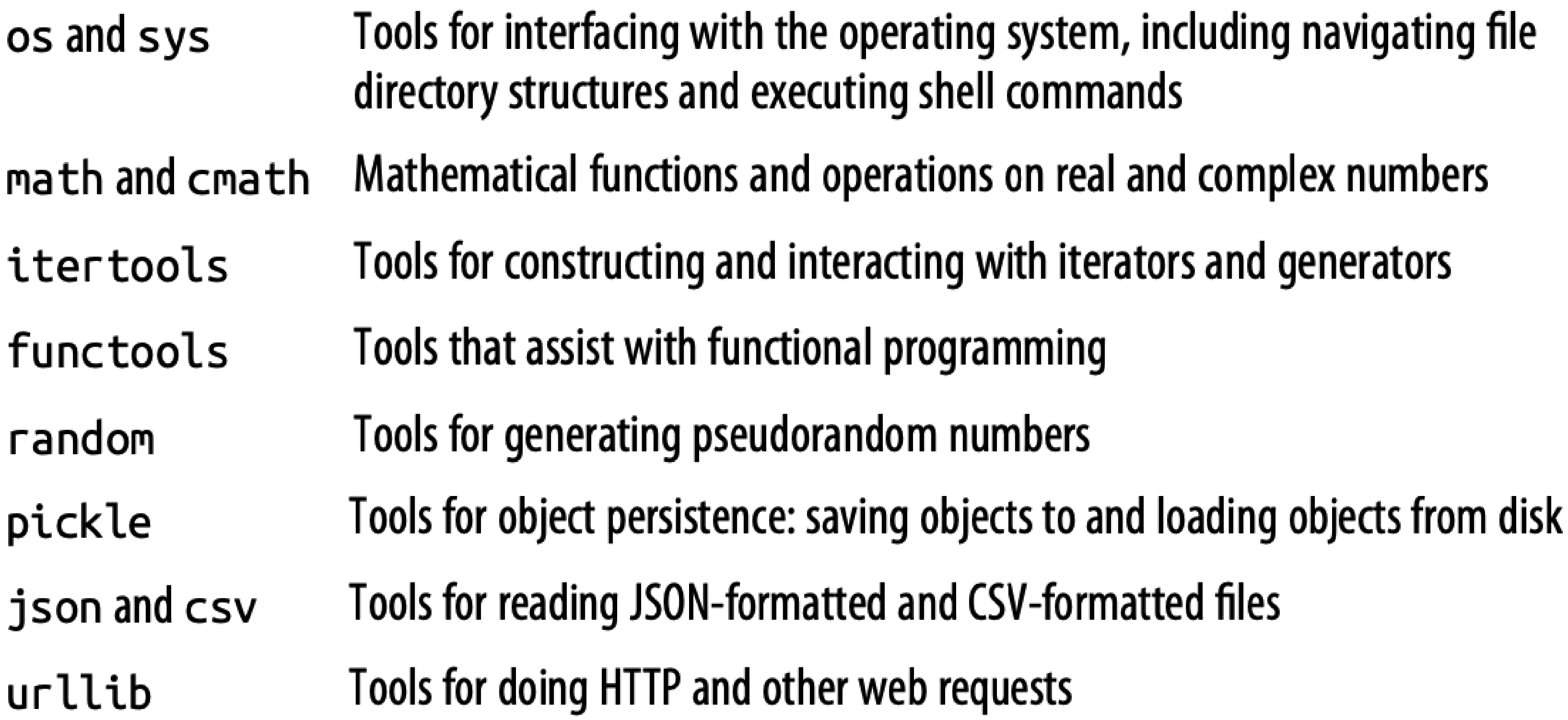
Implicit import of module contents



```
from math import *
sin(pi) ** 2 + cos(pi) ** 2
```

Loading Modules

Import from Python Standard Library

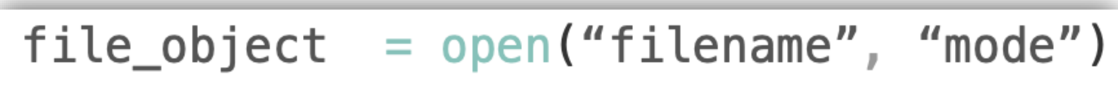


`os` and `sys`	Tools for interfacing with the operating system, including navigating file directory structures and executing shell commands
`math` and `cmath`	Mathematical functions and operations on real and complex numbers
`itertools`	Tools for constructing and interacting with iterators and generators
`functools`	Tools that assist with functional programming
`random`	Tools for generating pseudorandom numbers
`pickle`	Tools for object persistence: saving objects to and loading objects from disk
`json` and `csv`	Tools for reading JSON-formatted and CSV-formatted files
`urllib`	Tools for doing HTTP and other web requests

Files

Open ()

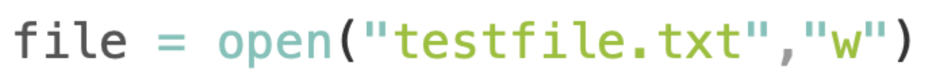
Opens a file object



```
file_object  = open(“filename”, “mode”)
```

Mode

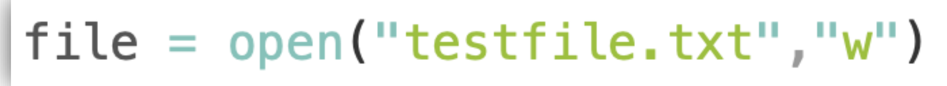
‘***r***’ – Read mode which is used when the file is only being read
‘***w***’ – Write mode which is used to edit and write new information to the file
‘***a***’ – Append mode, which is used to add new data to the end of the file
‘***r+***’ – Special read and write mode, which is used to handle both actions when working with a file



```
file = open("testfile.txt","w")
```

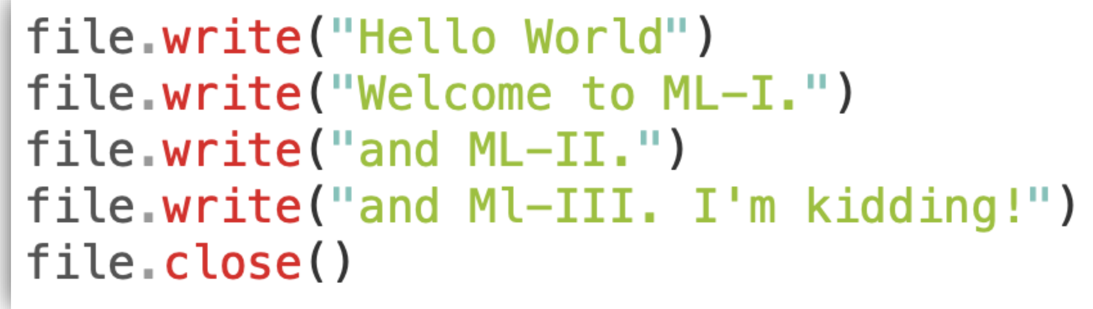
Creating a file

Create a file



```python
file = open("testfile.txt","w")
```

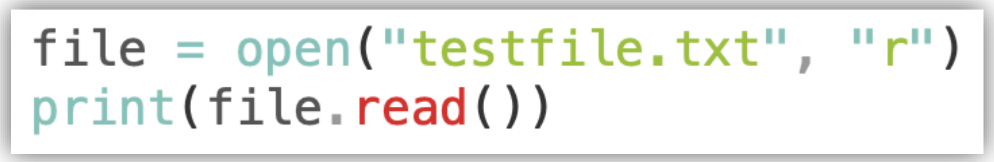
Write contents to a file



```python
file.write("Hello World")
file.write("Welcome to ML-I.")
file.write("and ML-II.")
file.write("and Ml-III. I'm kidding!")
file.close()
```

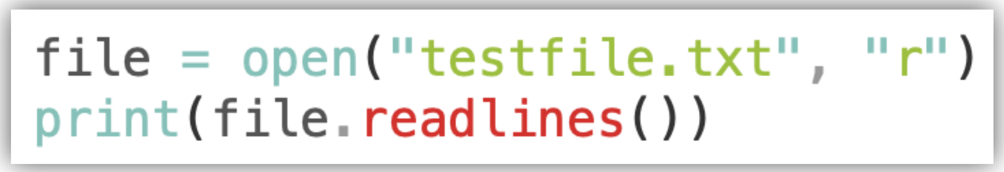
Reading a file

read



```
file = open("testfile.txt", "r")
print(file.read())
```

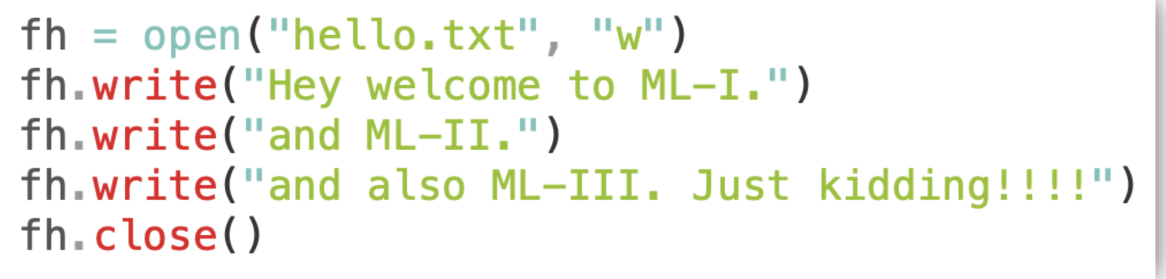
readlines



```
file = open("testfile.txt", "r")
print(file.readlines())
```

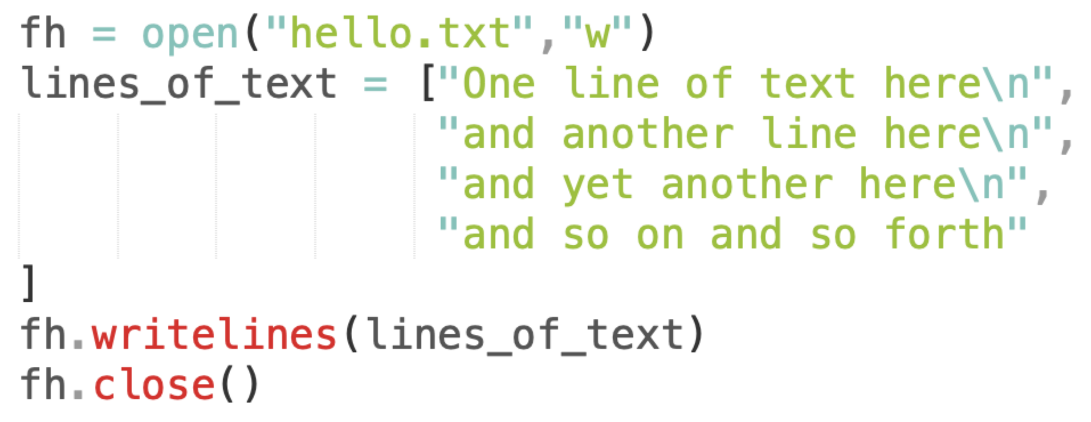
Writing a file

write



```
fh = open("hello.txt", "w")
fh.write("Hey welcome to ML-I.")
fh.write("and ML-II.")
fh.write("and also ML-III. Just kidding!!!!")
fh.close()
```

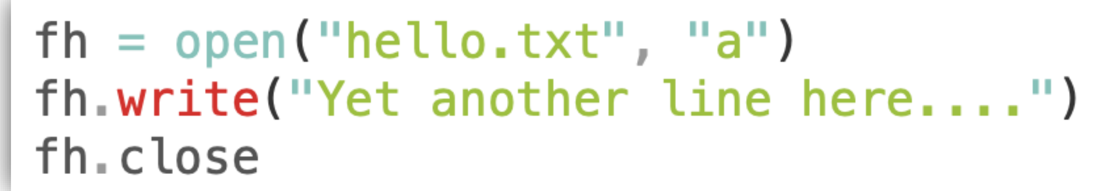
writelines



```
fh = open("hello.txt","w")
lines_of_text = ["One line of text here\n",
                 "and another line here\n",
                 "and yet another here\n",
                 "and so on and so forth"
]
fh.writelines(lines_of_text)
fh.close()
```

Appending a file

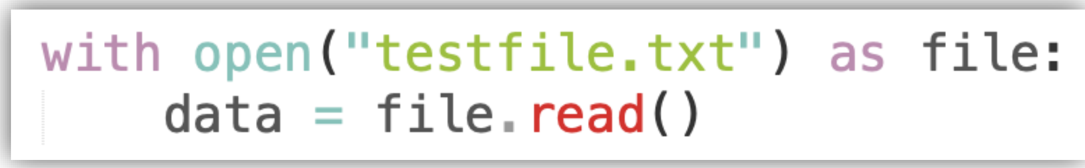
append



```
fh = open("hello.txt", "a")
fh.write("Yet another line here....")
fh.close
```

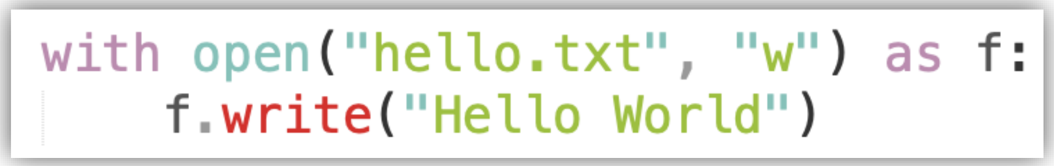
with

with read



```
with open("testfile.txt") as file:
    data = file.read()
```

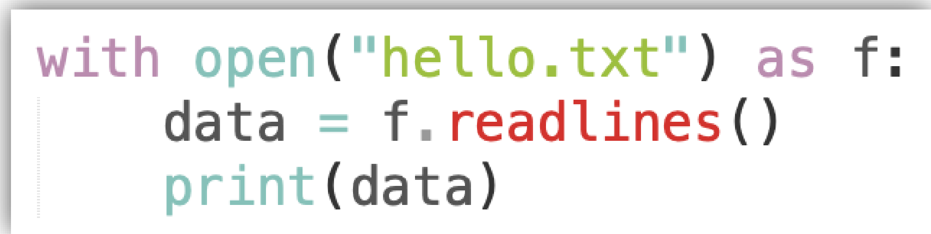
with write



```
with open("hello.txt", "w") as f:
    f.write("Hello World")
```

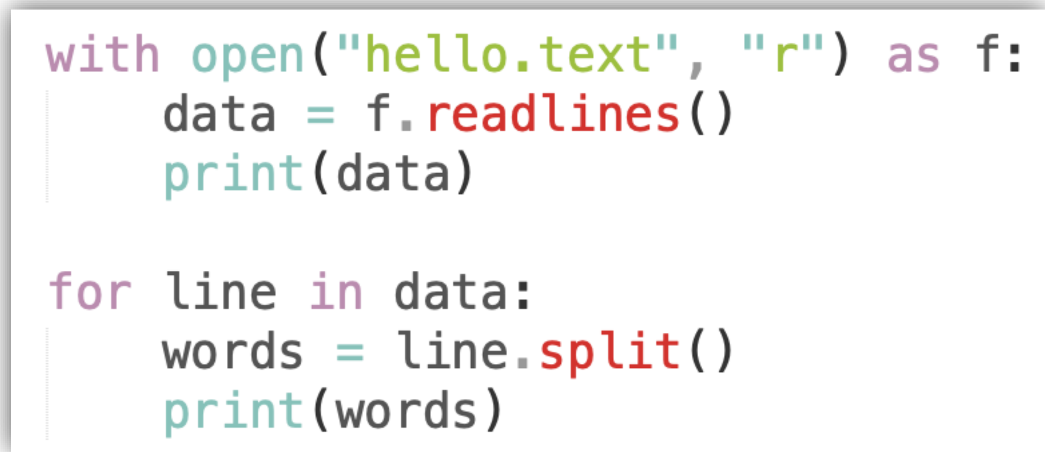
with

with readlines



```
with open("hello.txt") as f:
    data = f.readlines()
    print(data)
```

with splitting



```
with open("hello.text", "r") as f:
    data = f.readlines()
    print(data)

for line in data:
    words = line.split()
    print(words)
```

References & Further Reading

Images used in the deck is credited to these resources

- A whirlwind tour of Python *Jake Vanderplas*
- Learn Python the hard way *Zed Shaw*
- Python Crash Course: A Hands-On, Project-Based Introduction to Programming *Eric Matthes*

For more Python practice and get upto speed, look at the attached Python Notebooks for practice [Optional]

Jupyter Notebook

Jupyter Notebook

https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html

Let's take a tour

Task 1: Python script for Web Scraping

Task 1

Create a Python script for enabling web scraping.

Input: Url (medium article)

Output: File stored in your local machine with the text from the url

Hints: You require the following libraries to accomplish that.

os - File storing in the local directory of your machine

requests - Used for sending requests with url as input and get html source as response

beautifulSoup - Used in parsing the html source response and make sense

Import library

Study the library usage in its documentation

Fill in the blank provided in the Script

Submit the extracted and stored file containing the text under Assignment 1

Task 2: Push files in Github

Replicate Task 1 in a Jupyter Notebook and push file into Github account

Task 2

Replicate the answer from the previuos Python script for enabling web scraping into a Jupyter Notebook. Push these three files (text file, python script and notebook) into your Github account.
Input(s): Text (.txt), Python script (.py) and Notebook (.ipynb)
Output: Github account with these three files pushed, reviewed and merged into the master branch of your repo

Hints: You require the following to accomplish that.
python script - Task 1 answer (.py)
Text file - Task 1 text file (.txt)
Notebook - Replicated answer in a notebook file (.ipynb)
Github repo - Your repo that you have created already
Choose one of your classmate and add him/her as a collaborator in your Repo's settings
Commit, Push, Create a PR with a Reviewer
Wait for approval, Merge
Submit the repo link as the submission file for Task 2