Python

Who am I ?

Independent security researcher.

Specialized in low-level software Reverse Engineering.

Maintain several Python projects.

BRUNO PUJOS

Who Are You ?

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output
7. Exceptions
8. Object Oriented Programing
9. Modules
10. Project

Plan

Python

- General-purpose programming language
- High-level Language
- Interpreted
- Object Oriented
- Emphasizes on code readability

Python

Python History

December 1989

January 1994: Version 1.0

python

October 2000: Python2

December 2008: Python3

January 2020: EOL Python2

Python2 & Python3

- Not actually the same language
- Python3 is more consistent
- Everything we will see is compatible for both Python2 & Python3
- Some library & projects never got ported to Python3…

Python

Advantages

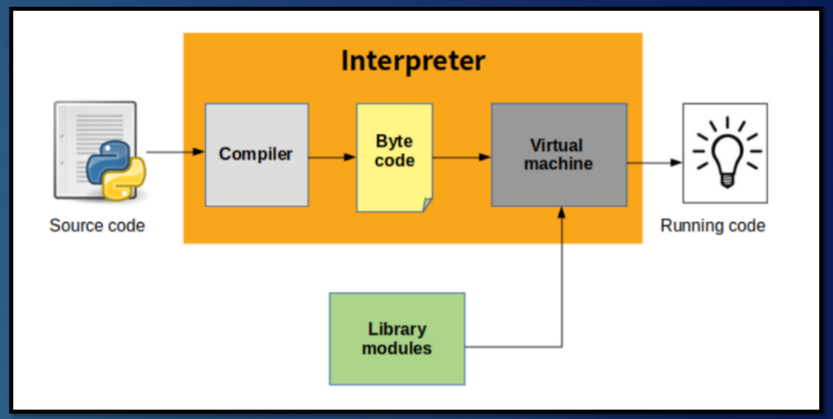
- Easy to read & write.
- Extensive library and supports.
- Use by a lot of people.
- Well supported and still evolving.

Weaknesses

- Slooooowwwwwwwww.
- Dynamically typed.
- You have to have the interpreter: it must be installed before using it.

Interpreter

- **Cpython**
  - Reference implementation
  - Open-source
  - Supported by the *Python Software Foundation*
  - Produce PYC files: contain the byte code.



- Can be used through:
  - Interactive interpreter.
  - Scripting.

Interpreter

- Read-Eval-Print Loop (REPL):
  1. Read: the command
  2. Evaluate & execute it
  3. Print the result
  4. Loop: restart at 1.

```
$ python3
Python 3.8.10 (default, Sep 28 2021, 16:10:42)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>>
```

Hello World

```
print("Hello World!")
```

Practice!

Print your name!

Steps:

1. Launch an interpreter (*python*)
2. Write the code for printing your name
3. Put the same code you write in *hello.py*
4. Launch the file (*python hello.py*)

Plan

Introduction **Python 101** **Types & Syntax**

Variables

- Variables are boxes
- Use for containing a value

```
>>> hello = "Hello"
>>> print(hello)
Hello
>>> hello = "Hello world!" # This is a comment
>>> print(hello)
Hello world!
>>> hello2 = "Good morning!"
>>> print(hello2)
Good morning!
>>> Hello = "Not the same as hello"
>>> print(Hello)
Not the same as hello
```

Data Types

Base Type

- String
- Integer
- Float
- Boolean (bool)

Testing and manipulating

- type
- isinstance

Basic Operations

- Integer:
  - Can make math operations: **+**, **-**, *, /, //, %
  - Or logical operation: |, **&**, **^**
- Float:
  - Same as integer.
  - What happens if I mix a float and an integer ?
- String
  - Concatenate with: **+**
  - What will happen with: * ?
- Boolean (bool)
  - Logical operation: **not**, **and**, **or**, **^**

Converting between types

- Python is **strongly**, **dynamically typed**.
- Change of type must be explicit.
- Can be done by calling the type constructor:

```
>>> a = 3
>>> a
3
>>> str(a)
'3'
>>> a = "5"
>>> a
'5'
>>> int(a)
5
>>> a = "5.1"
>>> float(a)
5.1
```

TypeError

- *TypeError* indicates that a problem occurs with the type
- Here it says we cannot concatenate integer and string
- This is called an **Exception**.

```
>>> a = 3
>>> print(a)
3
>>> a = "hello"
>>> print(a)
hello
>>> a = a + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

List

- list contains several values
- Indicated by **[]**

```
>>> l = [1, 2, 3]
>>> print(l)
[1, 2, 3]
>>> print(len(l))
3
>>> l += [4, 5]
>>> print(len(l))
5
>>> print(l)
[1, 2, 3, 4, 5]
>>> l2 = [1, "hello", 2, "world"]
>>> l2
[1, 'hello', 2, 'world']
>>> print(len(l2))
4
```

List & Index

- List elements can be accessed through index.
- Index start at 0.

```
>>> l = ["a", "b", "c"]
>>> l[0]
'a'
>>> l[1]
'b'
>>> l[2]
'c'
>>> l
['a', 'b', 'c']
>>> s = "xyz"
>>> s[0]
'x'
```

Conditions

- Program/Control flow: order in which the code is executed
- Can be manipulated and change using conditions.

```
green = True
red = False

if green: # First condition
    print("Cross the road")
elif red: # Only if first one was not true
    # You can put as many elif as you want
    print("Wait")
else:
    print("Look carefully before crossing")
```

Scope

- Define by *indentation* at the beginning of a line.
- Used for conditions, loops, functions, …
- Usually: 2 spaces, 4 spaces or a tabulation.
  - You can configure your IDE for taking that into account.
- Important to be consistent.

```
if green: # BAD
print("Cross the road")

if green: # GOOD
    print("Cross the road")
```

Loop

- Loops allow to repeat some code multiple time.
- **for** loops allow to iterate on a sequence:
  - list
  - string
  - …
- **while** loops allow to iterate until a condition is met.
- **break** allows to get out of a loop.
- **continue** allows to go to the next round.

```
>>> l = ["hello", "world"]
>>> for elt in l:
...     print(elt)
...
hello
world
```

Function

- Functions allow to repeat code.
- Can define a function with the **def** keyword.
- A function:
  - can take arguments **()**.
  - **return** values.

- Variable inside a function are local.
- Variables outside are global.

- Other possibilities:
  - Take named arguments with default values.
  - Take a variadic number of arguments.

```
def sum(arg1, arg2):
    end = arg1 + arg2
    print("Calculating sum: " + str(end))
    return end

result = sum(1, 4)
```

Keywords

- Keywords are reserved by the language
- Each of them has a specific meaning and usage
- Can't be used as names.

List of keywords

and exec not assert
finally or break
for pass class
from continue
global raise def
if return del
import try elif
in while else
is with except
lambda yield

Practice!

Types & Syntax!

Do all the exercises for Types & Syntax!

Plan

Introduction **Python 101** **Types & Syntax** **Standard Data Types**

Standard Data Types

- Integer
- Boolean
- String
- List
- Tuple
- Set
- Dictionary

Strings

```
str1 = 'Hello {var1} {var2} World!'
print(str1)
print(str1[0])
print(str1[2:5])
print(str1[2:])
print(str1[2:-2])
print(str1[:-3])
print(str1 * 2)
print(str1 + "TEST")
print(str1.format(var1='big', var2='big'))

str1.endswith("d!")
str1.endswith("d?")
str1.find("lo")
str1.endswith("la")
"llo" in str1
```

Methods & Attributes

- Methods are functions associated with an object
  - Called on the object
  - *obj.method()*
- Attributes are values associated with an object
  - *obj.attribute*
- You can use **dir** and **help** for having information about them.

List (1/2)

```
liste1 = ['abcd', 786 , 2.23, 'john', 70.2]
tinylist = [123, 'john']

print (liste1)
print(liste1[0])
print(liste1[1:3])
print(liste1[2:])
print(tinylist * 2)
print(liste1 + tinylist)
```

List (2/2)

```
a = []
a.append(4)
a.append(1)
a.sort()
print(a)
a.append('hello')
a.sort()
print(a)
b = [3,1]
a.extend(b)
a.reverse()
a.remove(1)
a.pop()
len(a)
"hello" in a
"hello" in b
3 in b
```

Slicing & Negative index

- Slicing:
  - obj[start:end:step]
  - if not precised, use the default:
    - start=0
    - end=len(obj)
    - step=1
- Negative index:
  - Start from the end of the list
  - Get the last elements: obj[-1]
- Can be combined:
  - Example: delete the last element of a list: list[:-1]

Tuple

- Sequences of elements
- Represented by **()**
- Tuples are **read-only**

```
tuple = ('abcd', 786 , 2.23, 'john', 70.2)
tinytuple = (123, 'john')

print(tuple)
print(tuple[0])
print(tuple[1:3])
print(tuple[2 :])
print(tinytuple * 2)
print(tuple + minusculetuple)

786 in tuple
"test" in tuple
```

Set

- Collection of unique data
- Not ordered
- Mutable
- Can't be used on all objects (but can on all basic one)

```
set = {1,3}
set.add(2)
set.update([2,3,4])
set.update([4,5],{1,6,8})
print(set)
set = {1,3,4,5,6}
set.discard(4)
set.remove(6)
set.remove(6)
```

Set

```
set1 = set([1,2,3])
list1 = list(set1)
set2 = {3,4,5}

set1 | set2 # union
set1 & set2 # intersection
set1 – set2 # difference1
set2 – set1 # difference2
set1 ^ set2 # symmetricdifference
```

Dictionary

- Collection of data
- Key-value association
- Unordered data
- Idea is that lists are dictionaries where the key is the index.

```
dict1 = {}
dict1['one'] = "Thisisone"
dict1[2] = "Thisistwo"

tinydict = {'name':'john','code':6734,
'dept':'sales'}

print(dict1['one'])
print(dict1[2])
print(tinydict)
print(tinydict.keys())
print(tinydict.values())
```

Dictionary

```
tinydict = {'name':'john','code':6734, 'dept':'sales'}

# test
"name" in tinydict
"john" in tinydict

# update & add
tinydict["name"] = "Ella"
tinydict["age"] = 20

# loop
for k in tinydict: # .items()
    print("{}: {}".format(k, tinydict[k]))

# deletion
del tinydict["name"]
tinydict.clear()
del tinydict
```

What happens when modifying a list in a function ?

Pass by reference

Variadic arguments

```
def printinfo(*args, **kargs):
    print("ARGS:")
    for a in args:
        print("    {}".format(a))
    print("KARGS:")
    for k, v in kargs.items():
        print("    {}: {}".format(k, v))

printinfo(3, 1, "test", hello="world")
```

Practice!

Standard Data Types!

Do all the exercises for Standard Data Types!

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation

Open

- For opening a file
- Arguments:
  - Path to your file
  - A mode
- Can be:
  - read
  - write
  - seek
  - close

Possible modes:

- r: read
- r+: read & write
- w: write, create if needed
- w+: read & write, create if needed
- a: append (write), create if needed
  - Position at the end
- a+: append & read
- b: binary (no encoding)

File Manipulation (1/2)

```
file = open('mylist.txt', 'r')

for ligne in file:
    print(ligne)
file.close()

with open('mylist.txt', 'r') as file:
    for ligne in file:
        print(ligne)

with open('mylist.txt', 'w') as file:
    file.write('hello\n')
    file.write('world\n')
    file.write('and galaxy!\n')
```

File Manipulation (2/2)

```
with open('mylist.txt', 'a') as file:
    file.write('It is ')
    file.write('really nice\n')
    file.write('to meet you!\n')

with open('mylist.txt', 'r') as file:
    print(file.name)
    print(file.closed)
    print(file.mode)

with open('mylist.txt', 'a+') as file:
    file.write("How are you ?\n")
    file.seek(0)
    print(file.read(5))
    file.seek(30)
    print(file.read(5))
```

OS module

- Modules are code repository
  - Allows to centralize code
  - Usually classed by category
  - Lost of common modules

- Module must be imported explicitly

```
import mymodule
mymodule.myfunc()
from mymodule import myfunc # from mymodule import *
myfunc()
```

- the **os** module allows to interface with the Operating System
  - in particular to manipulate files
  - the **os.path** module allows to manipulate file path
  - https://docs.python.org/3/library/os.html

OS File manipulation

```
import os
import os.path

os.rename("test1.txt", "test2.txt")
os.remove("text2.txt")
os.getcwd()
os.chdir("/tmp")
os.getcwd()
os.mkdir("test")
os.rmdir("/tmp/test")

os.path.basename("/tmp/test")
os.path.dirname("/tmp/test")
os.path.join("/tmp", "test")
```

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output

Standard Input/Output

```
# print and input for simple read/write
name = input("Enter your name: ")
print("Hello {}".format(name))

# sys.std* are like files
import sys
sys.stdin.read(10)
sys.stdin.readline()
sys.stdout.write("hello")
sys.stderr.write("hello")
```

Arguments & main script

```
import sys

if __name__ == "__main__":
    print(len(sys.argv))
    print(sys.argv)
else:
    print("Not launched directly")
```

Practice!

Files, Input & Output!

Do all the exercises for Files, input & outputs!

Practice!

Tic-tac-toe!

Make the tic-tac-toe mini-project.

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output
7. Exceptions

Exceptions

- Exceptions are errors which happen during program execution
- Those exceptions can be handled, using **try**, **catch** & **finally**.
- And you can trigger your own using **raise**.
- There is also different type of exceptions.
- Exceptions are **NOT** syntax errors.

```
print( 0 / 0)) # syntax error
print( 0 / 0) # Exception
```

```
try:
    open("donotexist.txt", "r")
except Exception as e:
    print("Got an exception")
    print(e)
    raise e
else: # not mandatory
    print("Did not got an exception")
finally: # not mandatory
    print("We do this all the time")
```

Exceptions


```
# your own exception
def mustbe5orless(x):
    if x > 5:
        raise Exception('Too big!')
mustbe5orless(10)

# AssertionError
import sys
def check_window():
    b = ('win' in sys.platform)
    assert b, "This code runs on Windows only."

try:
    check_window()
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
except AssertionError as error:
    print(error)
```

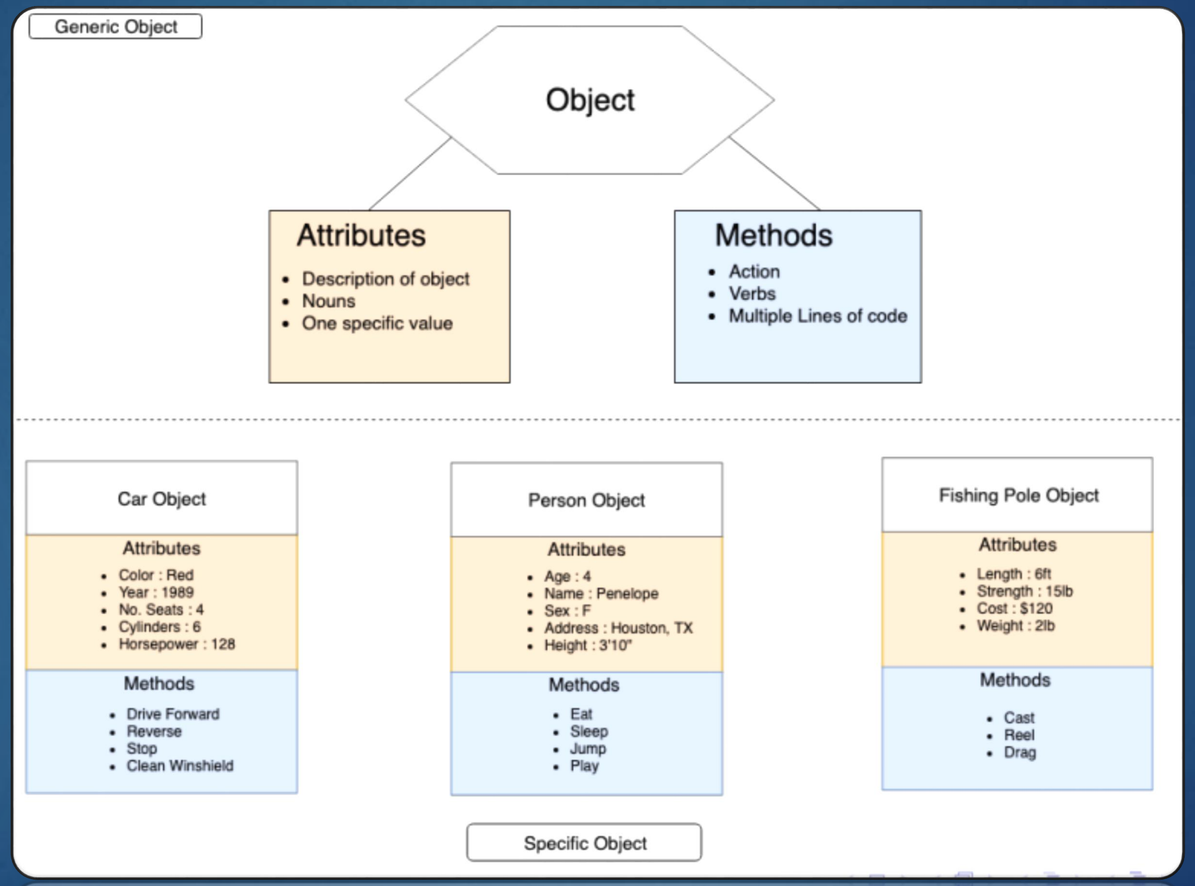
Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output
7. Exceptions
8. Object Oriented Programing

Objected Oriented Programming

- OOP is a programing paradigm.
- Based on **objects** which contains code and data.
  - Code is known as **methods** or **procedure**s.
  - Data is known as **fields** or **attributes** or **properties**.
- Usually based on the concept of **class**:
  - *Template which describes the creation, state and implementation behavior of an object.*
  - Most common ways to define objects.
  - Python use this concept.
- An object is an **instance** of a class.

Objected Oriented Programming

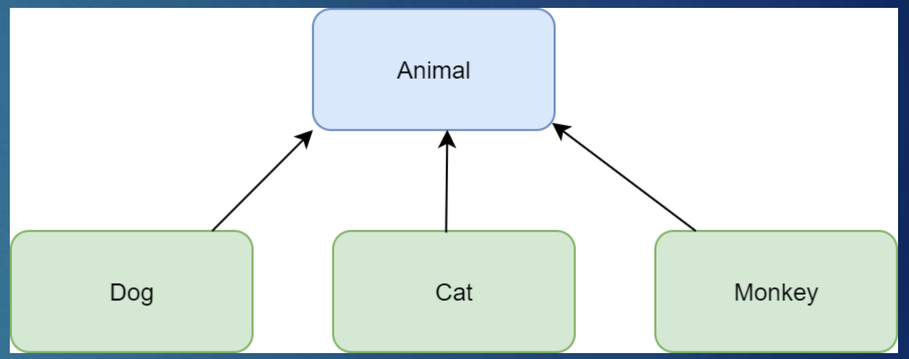


Class


```python
class ClassName(object):
    pass

class Employee:
    nb_employe = 0 # class variable

    def __init__(self, name, salary):
        self.name = name # instance variable
        self.salary = salary
        self.employe_id = Employee.nb_employe
        Employee.nb_employe += 1

    def display(self):
        print("Id: {}, Name: {}, Salary: {}".format(self.employe_id, self.name, self.salary))

print("NB employe: {}".format(Employee.nb_employe))
tom = Employee("Tom", 3000)
eve = Employee("Eve", 5000)
tom.display()
eve.display()
print("NB employe: {}".format(Employee.nb_employe))
```

Inheritance (1/2)

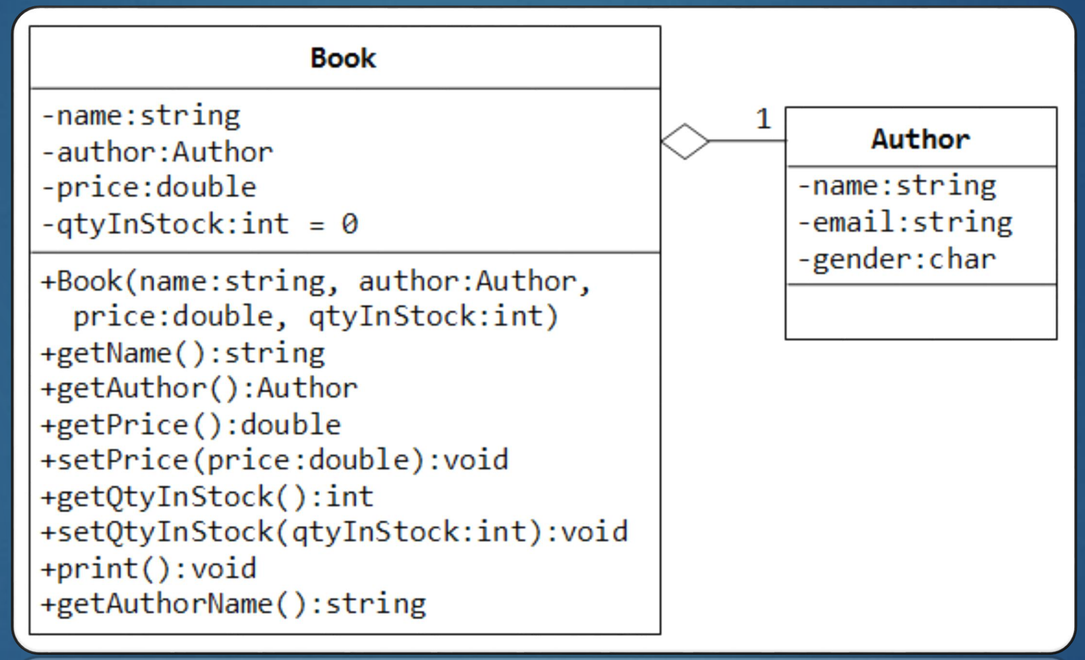
```
class Animal(object):

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

    def is_old(self):
        raise Exception("Depends!")

class Human(Animal):

    def __init__(self, name, age, job):
        super().__init__(name, age)
        self.job = job

    def is_old(self):
        return self.age > 100

class Dog(Animal):

    def is_old(self):
        return self.age > 20
```



Inheritance (2/2)

```
# V2
class Animal(object):
    OldAge = None

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

    def is_old(self):
        if self.OldAge is None:
            raise Exception("Depends!")
        return self.age > self.OldAge

class Human(Animal):
    OldAge = 100

    def __init__(self, name, age, job):
        super().__init__(name, age)
        self.job = job

class Dog(Animal):
    OldAge = 20
```

Classes together



Property

```
class Animal(object):
    OldAge = None

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

    @property
    def is_old(self):
        if self.OldAge is None:
            raise Exception("Depends!")
        return self.age > self.OldAge
```

Special methods

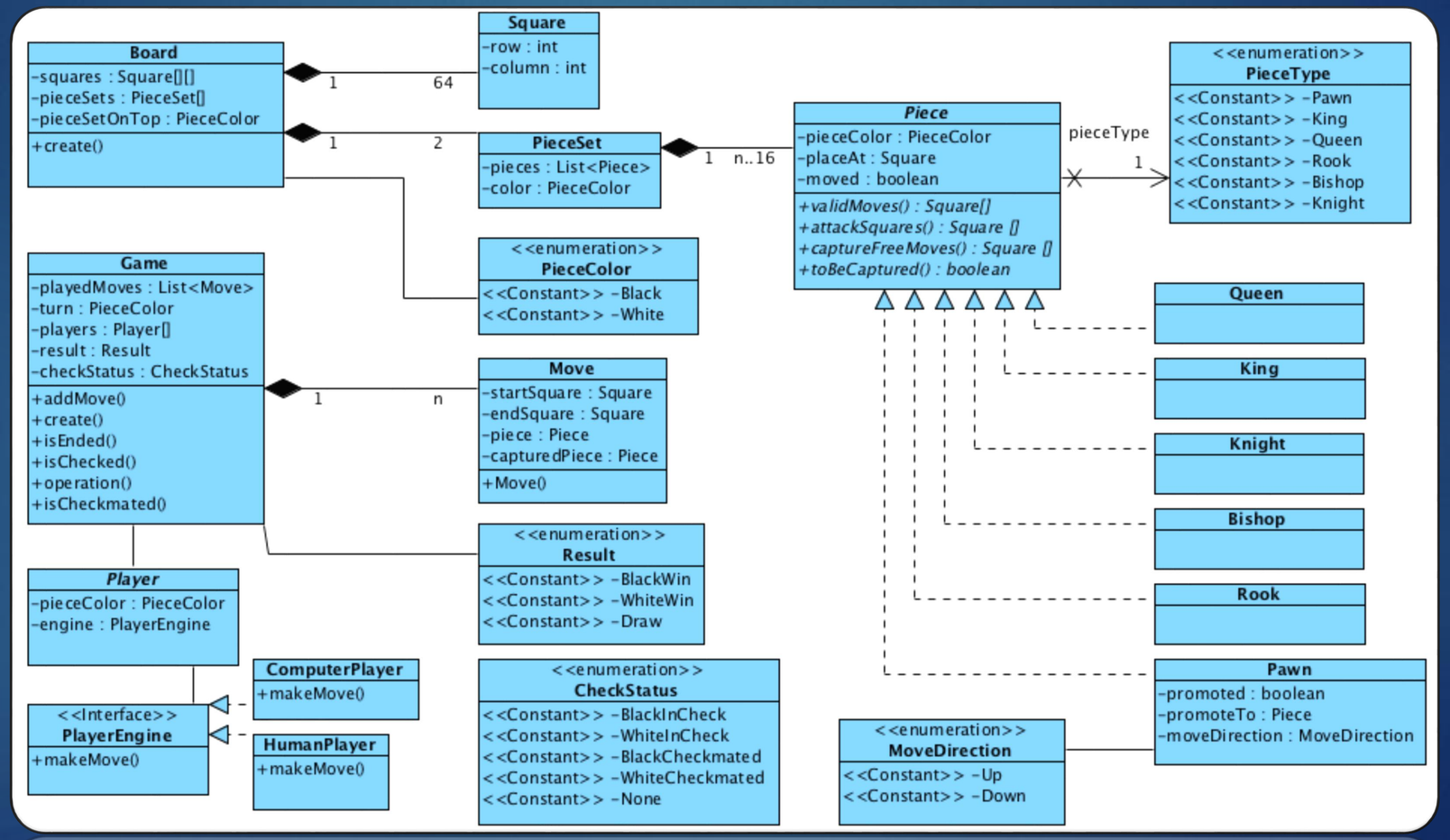
- Special methods for used by functions.

- __str__
- __add__, __sub__, __mul__, …
- __len__
- __iter__

```
class Animal(object):

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

    def __str__(self):
        return "Name: {}, Age: {}".format(self.name,
self.age)

    def display(self):
        print(str(self))

class Human(object)
    # ...

    def __str__(self):
        return super().__str__() + ", Job:
{}".format(self.job)
```

What classes are needed for representing a chess game ?

What classes are needed for representing a chess game ?



Practice!

Object Oriented Programing!

Do all the exercises for *Object Oriented Programing*!

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output
7. Exceptions
8. Object Oriented Programing
9. Modules

Modules

Modules

- "Package" of code
- Can contain class, functions, globals, …
- Can written in C or Python
- **dir** on a module for knowing what is in it

Common modules

- sys
- os & os.path
- json
- random
- request
- ctypes
- …

Module

```
import os
from os.path import join
from sys import *
import os.path as pa

dir(pa)
help(pa)
```

```
import sys
print(sys.path) # PYTHONPATH
```

Creating your module

- Possible to create your own module.
- Directory with a *__init__.py* file
  - That file can be empty
  - Code in it will be executed when imported.
- Interface exposed directly are the one included in the __init__.py
- Allows to create projects and split your code.

```
# mymodule/__init__.py
from .hello import hello_world

print("Hello from init!")
```

```
# mymodule/hello.py
def hello_world():
    print("Hello World!")

def private_hello():
    print("Private Hello")
```

Practice!

Modules!

Do all the exercises for *Modules*!

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output
7. Exceptions
8. Object Oriented Programing
9. Modules
10. Project

Project: Sudoku Solver

Plan

1. Introduction
2. Python 101
3. Types & Syntax
4. Standard Data Types
5. File Manipulation
6. Input & Output
7. Exceptions
8. Object Oriented Programing
9. Modules
10. Project

Documentation & Sphinx

THANK YOU

Bruno Pujos

Iterator & Generator

Hash

Decorator

Metaclass