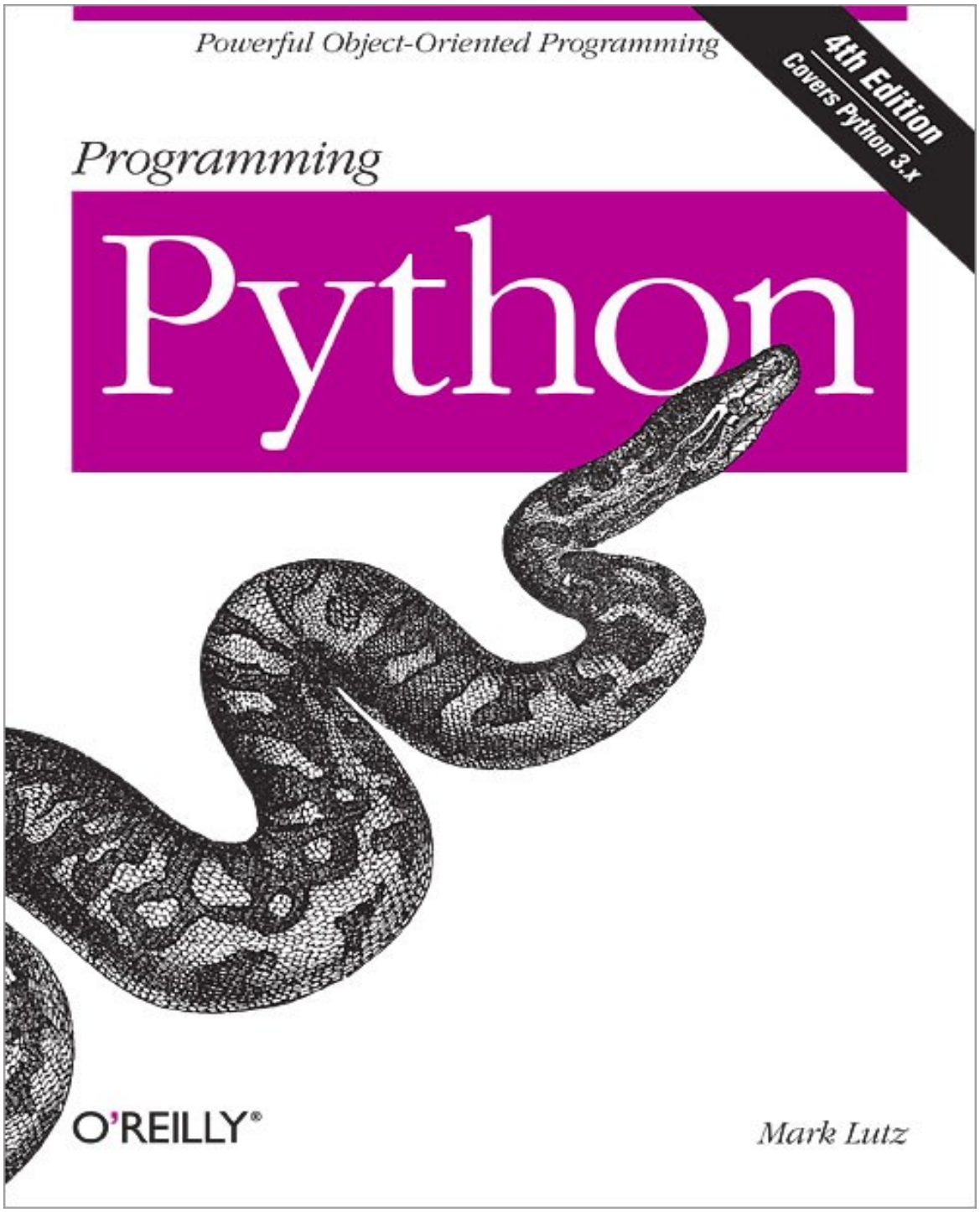

Powerful Object-Oriented Programming
4th Edition
Covers Python 3.x
Programming
Python
O'REILLY®
Mark Lutz

Functions

Functions

- Functions make programs easier to read, write and maintain

```
def say_hello():
    """Say hello."""
    print("Hello world!")
```

  - **Function definition:** Code that defines what a new function does
  - **Function header:** First line of a function definition
    - Give function name that conveys what it does or produces
  - **Docstring:** String that documents a function
    - Triple-quoted strings
    - Must be the first line in your function

Functions

- **Parameter:** A variable name inside the parentheses of a function header that can receive a value
- **Argument:** A value passed to a parameter

```
def say_hello(message):
    print(message)
```

- Parameters must get values; otherwise, error
- Multiple parameters can be listed, separated by commas

Default values

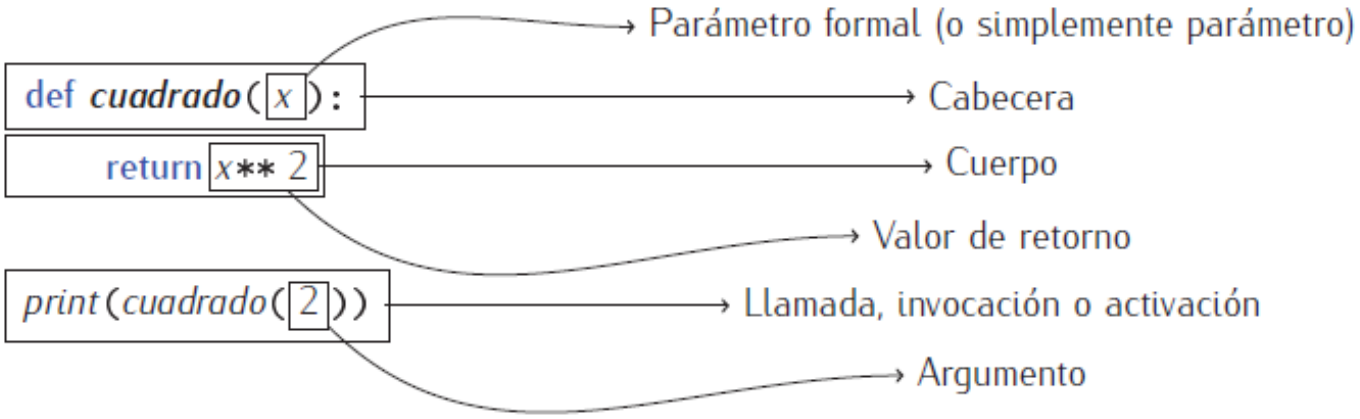
- We can define default values

```
def say_hello(message=”Hola mundo!”):
    print(message)
```

Functions

- **Return value:** A value returned by a function

  return statement returns values from a function and ends function call



- Can return more than one value from a function
  - list all the values in return statement, separated by

Recursive functions

- **Factorial :** x! = all the numbers from 1 to x multiplied together. 5! = 5 * 4 * 3 * 2 * 1

```
def factorial(x):
    F = 1
    for i in range(1, x+1):
        F *= i
    return F
```

Recursive functions

$$n! = n * (n-1)!$$

```
def factorial(num):
    if num==1:
        return 1
    else :
        return num * factorial(num-1)
```

Recursive functions

- In the Fibonacci sequence of numbers, each number is the sum of the previous two numbers, starting with 0 and 1

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,..

Recursive functions

```
def fib(n):
    fib_num = [1,1]
    for i in range(2,n):
        fib_num.append(fib_num[-1] + fib_num[-2])
    return fib_num[:n]
```

Recursive functions

- Fibonacci function:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n+2) = f(n) + f(n+1)$$

Recursive functions

```
def fibonacci(num):
        if num == 0:
                return 0
        elif num == 1:
                return 1
        else:
                return fibonacci(num - 1) + fibonacci(num - 2)
```

Functions as params

```
1 def suma(a, b):
2     return a + b
```

```
1 def calculadora(op, a, b):
2     return op(a,b)
```

```
1 calculadora(suma,5,7)
```

Lambda function

- # Here we have a function

```
def square_function(x):

    return x * x
```

- # equivalent lambda function

```
square = lambda x: x * x
```

Lambda function

```
1  div = lambda num, denom: num / denom if (denom != 0) else 0
```

```
1  def div(num, denom):
2      if (denom != 0):
3          return num / denom
4      else:
5          return 0
```

Functional programming

- Expression oriented functions of Python provides are:
  - list comprehension
  - lambda function
  - map(aFunction, aSequence)
  - filter(aFunction, aSequence)
  - reduce(aFunction, aSequence)

map

- map(function, sequence) function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.

```
1  list(map(lambda x: x**2, range(10)))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

filter

- filter(), requires the function to return boolean values (true or false) and then passes each element in the iterable through the function, "filtering" away those that are false. It has the following syntax:

```
1  scores = [66, 90, 68, 59, 76, 60, 88, 74, 91, 65]
2
3  def sobresaliente(score):
4      return score > 85
5
6  list(filter(sobresaliente, scores))
```

```
[90, 88, 91]
```

reduce

- reduce applies a function of two arguments cumulatively to the elements of an iterable, optionally starting with an initial argument. It has the following syntax:
- reduce(func, iterable[, initial])

```
1  from functools import reduce
2
3  numbers = [3, 4, 6, 9, 34, 12]
4
5  def custom_sum(first, second):
6      return first + second
7
8  reduce(custom_sum, numbers)
```

```
68
```

decorators

- decorators wrap a function, modifying its behavior.

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice

say_whee = do_twice(say_whee)
```

```
@do_twice
def say_whee():
    print("Whee!")
```

https://realpython.com/primer-on-python-decorators/#simple-decorators

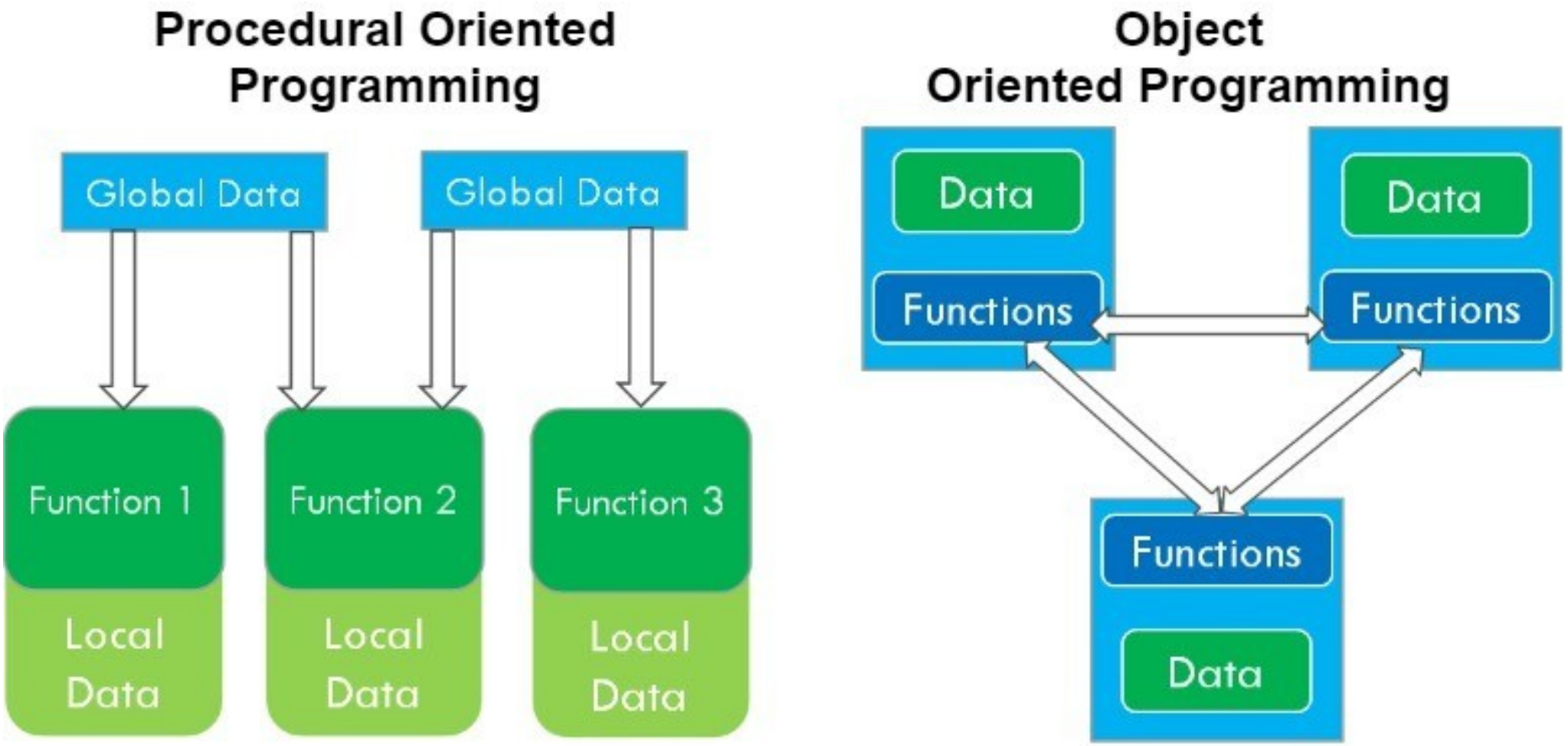
*args and **kwargs

- *args and **kwargs allow you to pass multiple arguments or keyword arguments to a function.

```
def my_function(a, b, *args, **kwargs):
    pass
```

https://realpython.com/python-kwargs-and-args/

Classes

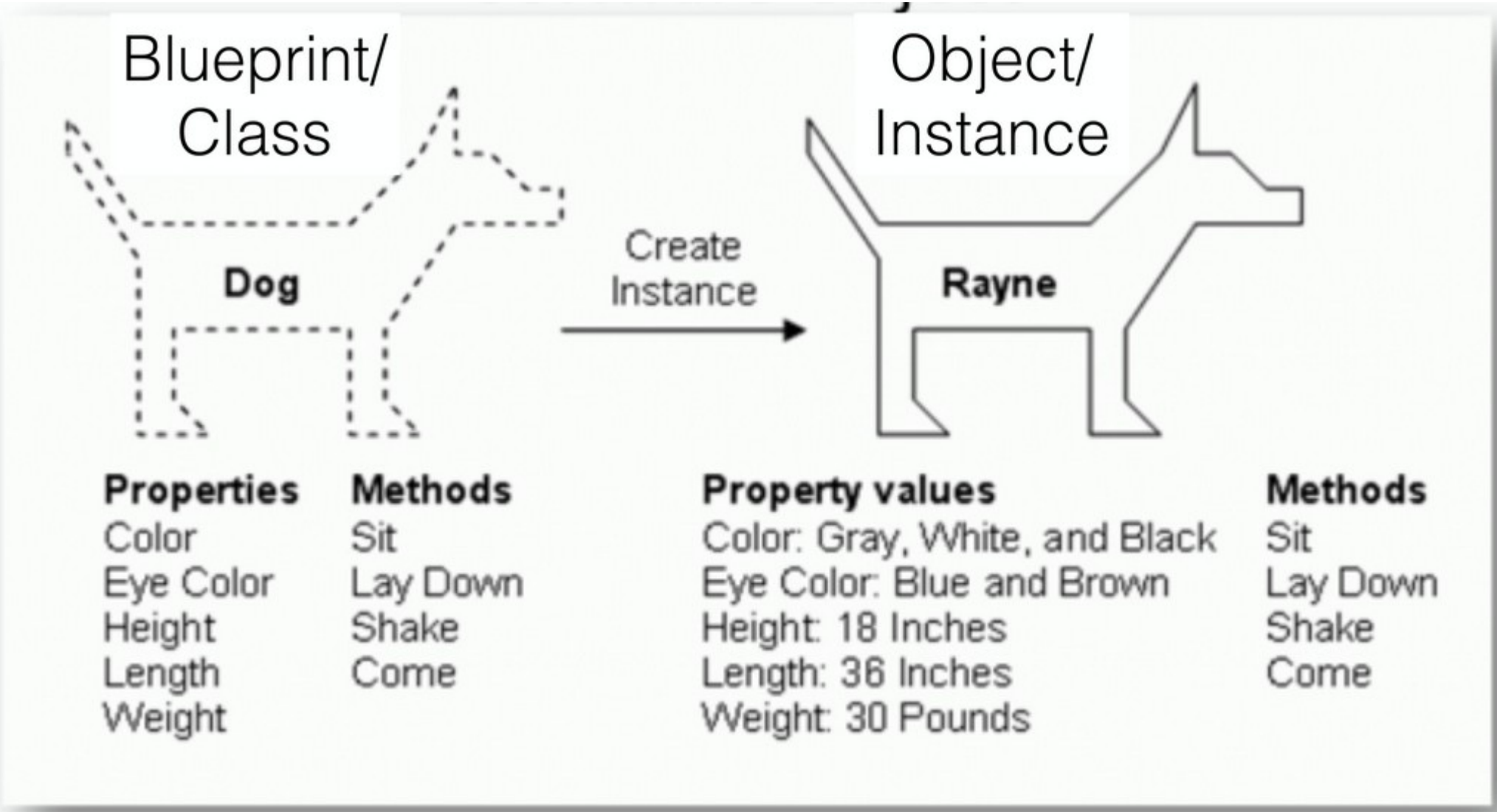

Procedural Oriented Programming
Global Data
Global Data
Function 1
Function 2
Function 3
Local Data
Local Data
Local Data
Object Oriented Programming
Data
Data
Functions
Functions
Functions
Data

Dog Properties

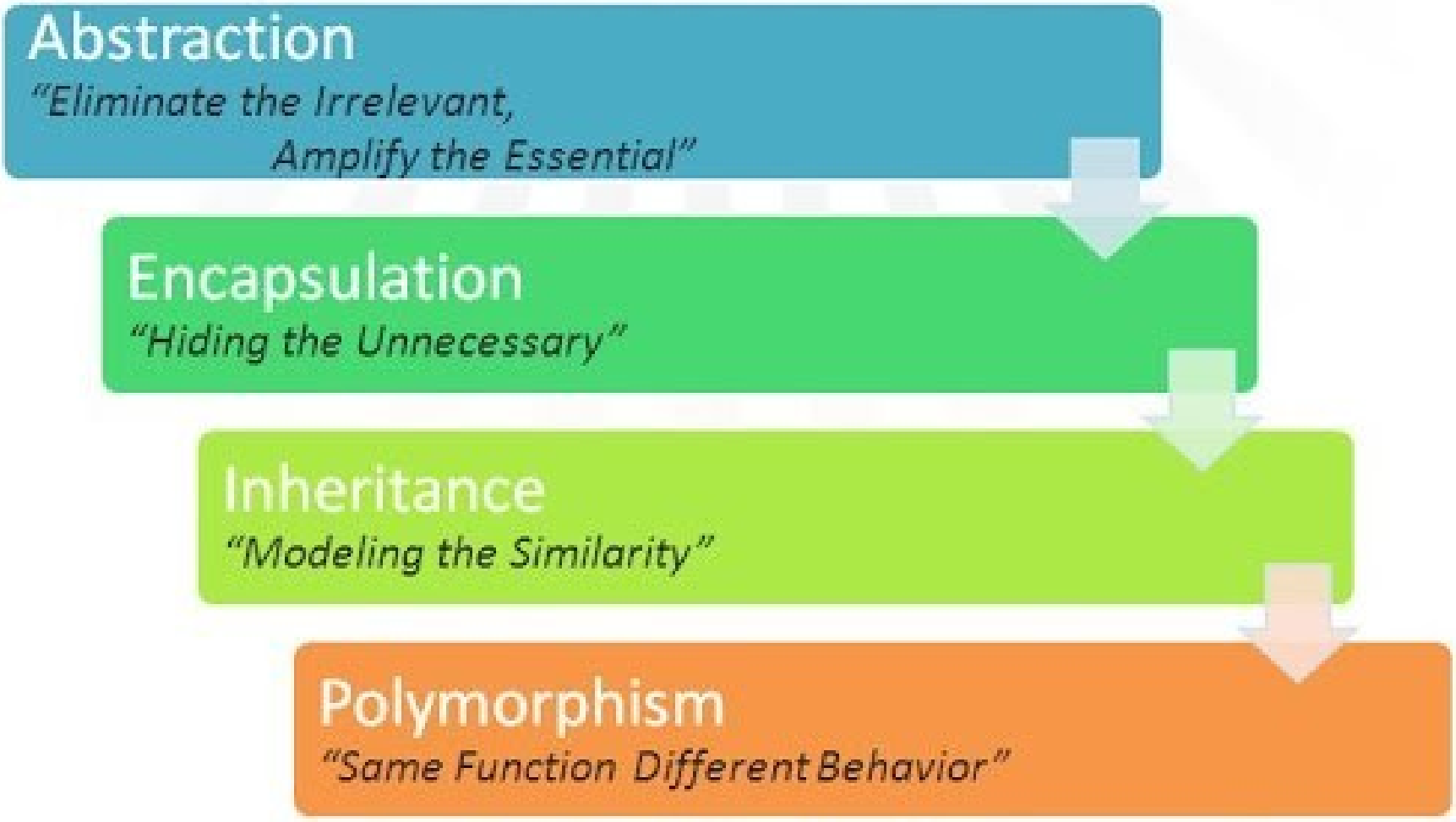
Color
Eye Color
Height
Length
Weight

Dog Behavior

Bark
Sit
Lay Down
Shake
Come


Blueprint/
Class
Object/
Instance
Dog
Create
Instance
Rayne
Properties
Color
Eye Color
Height
Length
Weight
Methods
Sit
Lay Down
Shake
Come
Property values
Color: Gray, White, and Black
Eye Color: Blue and Brown
Height: 18 Inches
Length: 36 Inches
Weight: 30 Pounds
Methods
Sit
Lay Down
Shake
Come

Object Oriented Programming



Classes

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Classes

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

- MyClass.i
- MyClass.f()
- MyClass.__doc__
- X=new MyClass()
  - Crea una instancia de la clase MyClass y la guarda en la variable X del tipo objeto

Classes

- Constructor de objetos en python

  def __init__(self):

    self.data = []

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Classes

Los atributos, al igual que las variables, no necesitan ser declarados y simplemente son creados al ser utilizados

```
x = new MyClass()
x.counter = 1


while x.counter < 10:
    x.counter = x.counter * 2
    print(x.counter)
```

Classes

```
def Persona():
    def __init__(self, nombre=None):
        self.nombre = nombre
    def saludo(self):
        If self.nombre:
            print('Hola, me llamo {}'.format(self.nombre))
            tu_nombre = raw_input('Cómo te llamas? ')
            print('Hola,', tu_nombre)


x = new Persona()

x.saludo()
```

Classes

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Classes

- objeto.__class__
- isinstance()
- isinstance(obj, int) es verdadero si obj.__class__ es int o una clase derivada de int.
- issubclass()
- issubclass(bool, int) es verdadero porque bool es una subclase de int.

Iterators and generators

Iterators

- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

- All these objects have a iter() method which is used to get an iterator:

```
myit = iter(mytuple)

print(next(myit))
```

https://www.w3schools.com/python/python_iterators.asp

Iterators

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)
```

generator functions

- generator functions are a special kind of function that return a lazy iterator. These are objects that you can loop over like a list. However, unlike lists, lazy iterators do not store their contents in memory.

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

```
def csv_reader(file_name):
    for row in open(file_name, "r"):
        yield row
```

https://realpython.com/introduction-to-python-generators/

generator functions

- yield indicates where a value is sent back to the caller, but unlike return, you don't exit the function afterward.
- Instead, the state of the function is remembered. That way, when next() is called on a generator object (either explicitly or implicitly within a for loop), the previously yielded variable num is incremented, and then yielded again.

Random numbers

random

```
>>> random()                    # Random float:  0.0 <= x < 1.0
0.37444887175646646
>>> randrange(10)                 # Integer from 0 to 9 inclusive
7
>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'
>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                 # Shuffle a list
['four', 'two', 'ace', 'three']
>>> sample([10, 20, 30, 40, 50], k=4)    # Four samples without replacement
[40, 10, 50, 30]
```

Exceptions

Exceptions

```
>>> 1/0

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    1/0
```

- **ZeroDivisionError**: integer division or modulo by zero
- **Exception**: An error that occurs during the execution of a program
- Exception is **raised** and can be **caught** (or trapped) then **handled**
- Unhandled, **halts** program and error message displayed

Exceptions

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly

>>> f = open("archivo.txt","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'archivo.txt'
```

Exceptions

- **SyntaxError**: When code has been typed incorrectly.
- **AttributeError**: When you try to access an attribute on an object that does not exist.
- **KeyError**: When you try to access a key in a dictionary that does not exist.
- **TypeError**: When an argument to a function is not of the right type (e.g. a str instead of int).
- **ValueError**: When an argument to a function is of the right type but is not in the right domain (e.g. an empty string)
- **ImportError**: When an import fails.
- **IOError**: When Python cannot access a file correctly on disk.

Catching Exceptions

```
try:
    num = float(input("Enter a number: "))
except:
    print("Something went wrong!")
```

- **try** statement sections off code that could raise exception
- Instead of raising exception, **except** block run
- If no exception raised, except block skipped

Catching Exceptions

```
try:
    num = float(input("Enter a number: "))
except(ValueError):
    print("That was not a number!")
```

- Different types of errors raise **different types of exceptions**
- **except** clause can specify exception types to handle
- Attempt to convert "Hi!" to float raises ValueError exception
- Good programming practice to specify exception types to handle each individual case
- Avoid general, catch-all exception handling

Catching Exceptions

```
try:
  num = float(input("\nEnter a number: "))
except ValueError as e:
  print("Not a number! Or as Python would say".format(e))
```

- Exception may have an argument, usually message describing exception
- Get the argument if a variable is listed before the colon in except statement

Catching Exceptions

```
for value in (None, "Hi!", 5):
    try:
      print("Attempting to convert {} –>{}".
                format(value, float(value)))
    except(TypeError, ValueError):
       print "Something went wrong!"
```

- Can trap for multiple exception types
- Can list different exception types in a single except clause
- Code will catch either TypeError or ValueError exceptions

Catching Exceptions

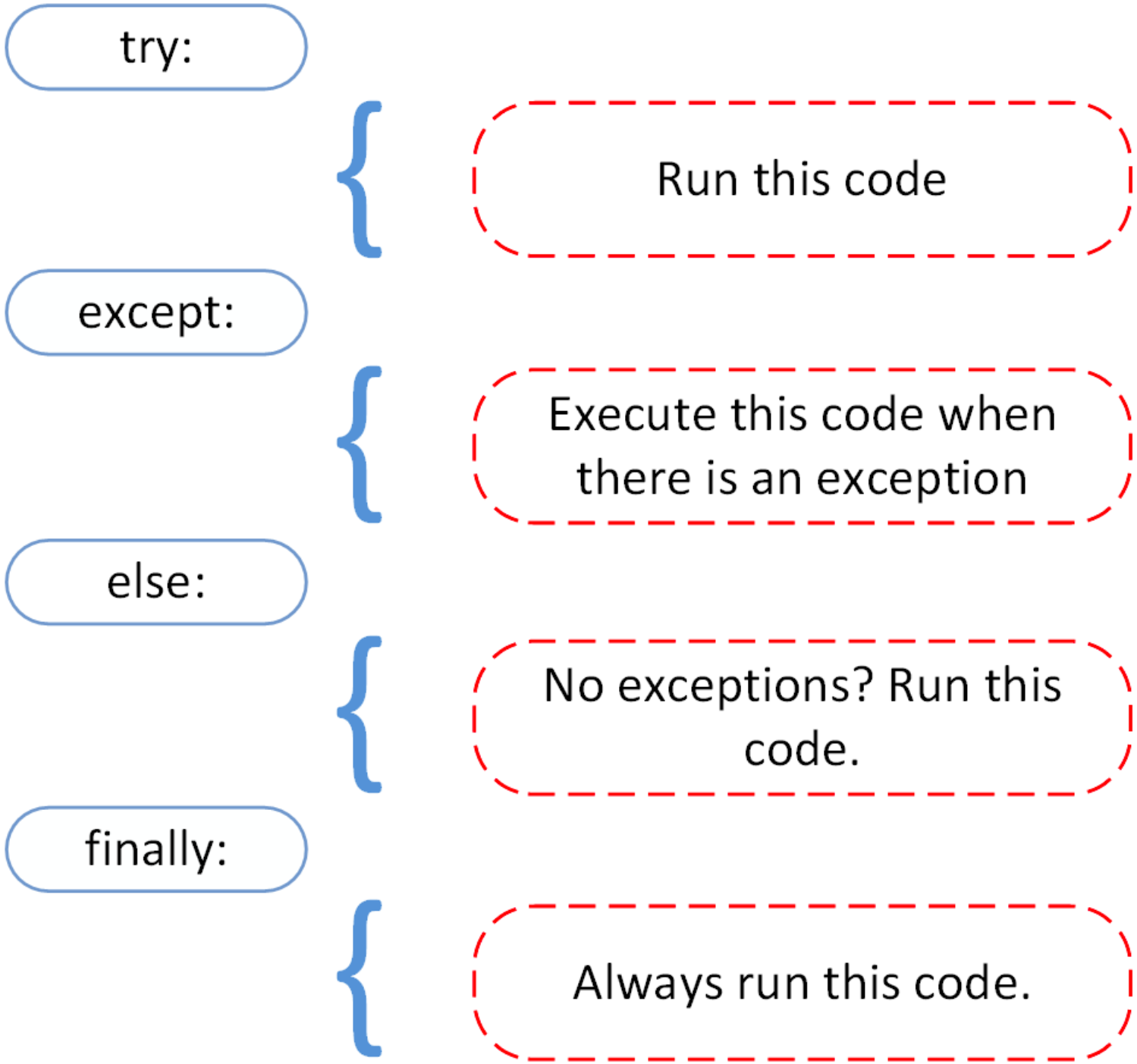
```
for value in (None, "Hi!", 5):
  try:
     print("Attempting to convert {} –>{}".
                format(value, float(value)))
  except(TypeError):
     print("Can only convert string or number!")
  except(ValueError):
     print("Can only convert a string of digits!")
```

- Another method to trap for multiple exception types is multiple except clauses after single try
- Each except clause can offer specific code for each individual exception type

Catching Exceptions

```
try:
    num = float(input("\nEnter a number: "))
except(ValueError):
    print("That was not a number!")
else:
    print("You entered the number {}".format(num))
```

- Can add single **else** clause after all except clauses
- **else** block executes only if no exception is raised
- num printed only if assignment statement in the try block raises no exception


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Files

Input/Output

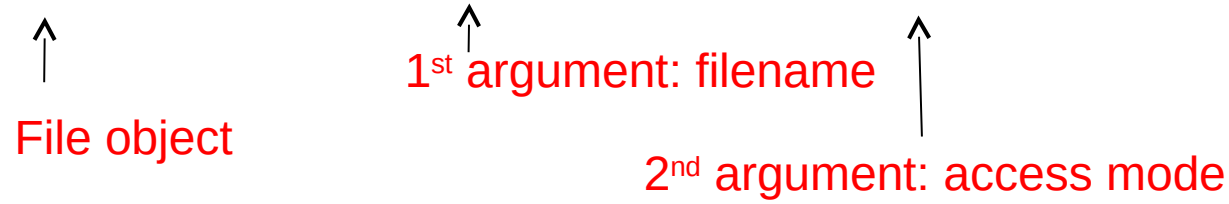
- So far we know how to get user's input using **input()**, and print out to the screen using **print()** statements!
- Now we are going to learn how to use files
- Read from text files
- Write to text files (permanent storage)
- Need to open a file before using it, and close it when it is done

Text files

- Plain text file: File made up of only ASCII/Unicode characters
- Easy to read strings from plain text files
- Text files good choice for simple information
  - Easy to edit
  - Cross-platform
  - Human readable!

Open files

```
myfile = open("read_it.txt", "r")
```




```
myfile.close()
```

- Must open before read or write; then you read from and/or write to the file by referring to the file object
- Always close file when done reading or writing

Access Modes

TABLE 7.1 SELECTED FILE ACCESS MODES

Mode	Description
`"r"`	Read from a file. If the file doesn't exist, Python will complain with an error.
`"w"`	Write to a file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
`"a"`	Append a file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.
`"r+"`	Read from and write to a file. If the file doesn't exist, Python will complain with an error.
`"w+"`	Write to and read from a file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
`"a+"`	Append and read from a file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.

Reading files

```
oneletter = text_file.read(1)    #read one character
fiveletter = text_file.read(5)     #read 5 characters
whole_thing = text_file.read()  #read the entire file
```

- **read()** file object method
- Argument: number of characters to be read; if not given, get the entire file
- Return value: string

- Each **read()** begins where the last ended
- At end of file, **read()** returns empty string

Reading files

```
text_file = open("read_it.txt", "r")


line1 = text_file.readline()
line2 = text_file.readline()
line3 = text_file.readline()
```

- **readline()** file object method
- Returns the entire line if no value passed
- Once read all of the characters of a line (including the newline), next line becomes current line

Reading files

```
text_file = open("read_it.txt", "r")
lines = text_file.readlines()
```

- lines is a list!
- **readlines()** file object method
- Reads text file into a list
- Returns list of strings
- Each line of file becomes a string element in list
- Compared to: read(), which reads the entire file into a string (instead of a list of strings)

Reading files

- Can iterate over open text file, one line at a time

```
>>> text_file = open("read_it.txt", "r")
>>> for line in text_file:
        print(line)
Line 1
This is line 2
That makes this line 3
```

Reading data (CSV, etc)

```
text_file = open("read_it.txt", "r")
for line in text_file:
    line = line.strip()
    (name, score) = line.split(",")
```

- **str.split**([sep[, maxsplit]]) -- Return a list of the words in the string, using sep as the delimiter string. If sep is not specified or None, any whitespace string is a separator '1<>2<>3'.split('<>') returns ['1', '2', '3'])
- **str.strip**([chars]) -- Return a copy of the string with the leading and trailing characters removed'   spacious   '.strip() returns 'spacious'

Writing files

- **write()** file object method writes characters

```
text_file = open("write_it.txt", "w")
text_file.write("Line 1\n")
text_file.write("This is line 2\n")
```

- **writelines()** writes list of strings to a file

```
text_file = open("write_it.txt", "w")
lines = ["Line 1\n", "This is line 2\n", "That makes this line 3\n"]
text_file.writelines(lines)
```

Writing files

```
with open("read_it.txt", "w") as text_file:
    lines = ["Line 1\n", "This is line 2\n", "That makes this line 3\n"]
    for line in lines:
        print(line, file=text_file)
```

os module

- The **os** module provides functions for interacting with the operating system (files and directories)
  - os.getcwd() # Return the current working directory
  - os.chdir() #change the working directory
  - os.path.exists('/usr/local/bin/python')
  - os.path.isfile(‘test.txt’)
  - os.listdir(os.getcwd()) #get a list of files in current directory

pathlib

- A new library for dealing with file paths:
  - https://zetcode.com/python/pathlib/

Path values

- p.name final path component.
- p.stem final path component without suffix.
- p.suffix suffix.
- p.as_posix() string representation with forward slashes (/):
- p.resolve() resolves symbolic links and “..”
- p.as_uri() path as a file URI: file://a/b/c.txt
- p.parts a tuple of path components.
- p.drive Windows drive from path.
- p.root root of directory structure.

Path values

- `Path.cwd()` current working directory.
- `Path.home()`User home directory

Path properties

- p.is_absolute() Checks whether the path is not relative.
- p.exists() Does a file or directory exist.
- os.path.abspath(path) Absolute version of a relative path.
- os.path.commonpath(paths) Longest common sub-path.
- p.stat() Info about path (.st_size; .st_mtime)
- p.is_dir() True if directory.
- p.is_file() True if file.

Listing subdirectories

- import pathlib

- p = pathlib.Path('.')
- for x in p.iterdir():
- if x.is_dir():
- print(x)

Open files

- path.read() A variety of methods for reading files as an entire object, rather than parsing it.
-
- with path.open() as f:
- lines = f.readlines()
- print(lines)

Path manipulation

- p.rename(target) Rename top file or directory to target.
- p.with_name(name) Returns new path with changed filename.
- p.with_suffix(suffix) Returns new path with the file extension changed.
- p.rmdir() Remove directory; must be empty.
- p.touch(mode=0o666, exist_ok=True) "Touch" file; i.e. make empty file.
- p.mkdir(mode=0o666, parents=False, exist_ok=False) Make directory.
- If parents=True any missing parent directories will be created.
- exist_ok controls error raising.

pathlib

- path = Path('words.txt')
- content = path.read_text()
- print(content)

- path = Path('copia.txt')
- path.touch()
- path.write_text('This is myfile.txt')

Example

```
#!/usr/bin/env python

from pathlib import Path
import datetime

now = datetime.datetime.now()
year = now.year
month = now.month

name = input('Enter article name:')

path1 = Path('articles') / str(year) / str(month)
path1.mkdir(parents=True)

path2 = path1 /  f'{name}.txt'

path2.touch()

print(f'Article created at: {path2}')
```


File operations
Play with file path
relative & absolute path
os.path.abspath()
os.path.relpath()
pathlib.Path().absolute()
pathlib.Path()
join paths
os.path.join("/home","tst.txt")
pathlib.Path("/home") / "tst.txt"
find parent directory
os.path.dirname()
os.pathlib.dirname(os.path.abspath())
pathlib.Path().parent
pathlib.Path().resolve().parent
Open & Close
f=open('tst.txt','w+') f.close()
with open('tst.txt','w+') as f:
Read
read()
readline()
readlines()
Write
write()
writelines()
json writer
csv writer
Search file
glob.glob()
os.endswith()
os.startswith()
pathlib.glob()
Move file pointer
seek()
tell()
File status
os.stat('tst.txt')
pathlib.Path('tst.txt').stat()
Copy, Move & Delete
shutil.copy()
shutil.copy2()
shutil.copyfile()
shutil.move()
os.system()
os.rename()
os.remove()
async solution with threading/subprocess