Outline

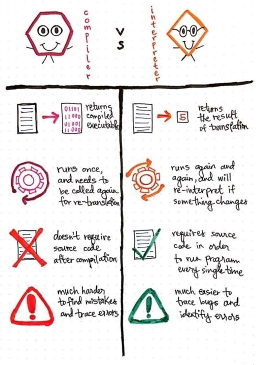
- Getting Started
  - About Python
  - Python Versions
  - Installation & Running
- The Basics
  - Identifiers and Keywords
  - Operators
  - Control structure
  - Reference Semantics
  - Data Structures
  - …

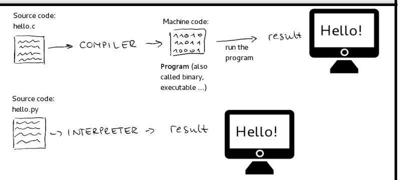
Python

"The best things in life are free!"

- Python is a dynamic programming language
  - Open source
  - High-level
  - Object-oriented
  - Interpreted
  - General-purpose
- Its coding structure enables programmers to articulate computing concepts in fewer lines of code
  - Simpler than C++ and Java

Python is Interpreted


compiler
vs
interpreter
returns compiled executable
returns the result of translation
runs once, and needs to be called again for re-translation
runs again and again, and will re-interpret if something changes
doesn't require source code after compilation
requires source code in order to run program every single time
much harder to find mistakes and trace errors
much easier to trace bugs and identify errors



Compiled		Interpreted	
PROS	CONS	PROS	CONS
ready to run	**not** cross platform	cross-platform	interpreter required
often **faster**	inflexible	simpler to test	often **slower**
source code is **private**	extra step	easier to debug	source code is **public**

- **Cython** is an **optimising static compiler** for both the Python programming language and the extended Cython programming language

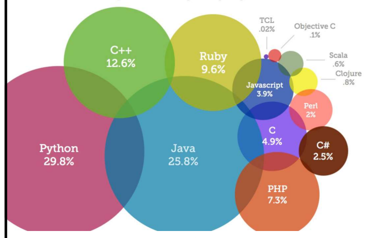
…some "mottos" of Python

- Beautiful is better than ugly → be consistent!
- Complex is better than complicated → use existing libraries!
- Simple is better than complex → Keep It Simple and Stupid (KISS)!
- Flat is better than nested → avoid nested "ifs"!
- Explicit is better than implicit → be clear!
- Sparse is better than dense → separate code into modules!
- Readability counts → indenting for easy readability!
- Special cases aren't special enough to break the rules → everything is an object!
- Errors should never pass silently → good exception handler!

Python: The Rising Star

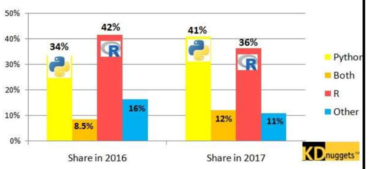
- Python was officially born on February 20, 1991, with version number 0.9.0
  - It has taken a tremendous growth path

Most Popular Coding Languages of 2018



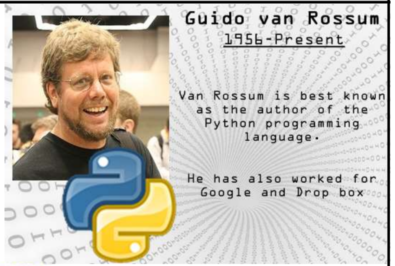
source: CRIPSY$_{Code}$

Python, R, Both, or Other platforms for Analytics, Data Science, Machine Learning



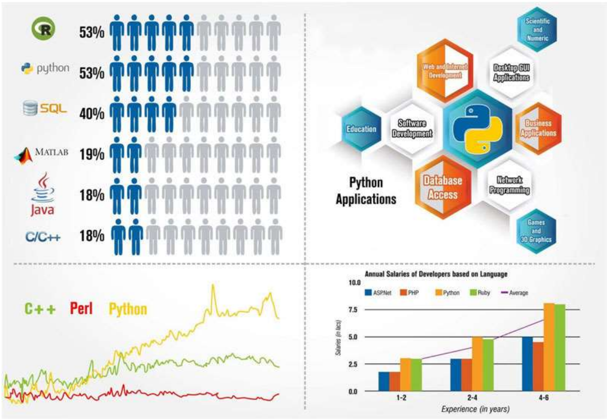
Story behind the name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus"


Guido van Rossum
1956-Present
Van Rossum is best known as the author of the Python programming language.
He has also worked for Google and Drop box

He doesn't particularly like snakes!!!

Python Applications



source: Cognixia

Python Versions

- 3.x is the latest version of Python
  - Pros: it has nicer and consistent functionalities
  - Cons: not all third-party modules support it
- 2.x is the most used version
  - Many frameworks still run on this version
- There are some differences into the code syntax
  - print-syntax
  - integer-division
  - Unicode
  - …

Installation & Running (1)

- [Windows]
  - Visit https://www.python.org/downloads/ and download the latest version
    - Note that if your Windows version is pre-Vista, you should download Python 3.4
  - During the installation make sure you check option Add Python 3.6 to PATH
    - Alternatively, define an environmental variable manually
    - `PATH → Add → C:\[user-dir]\[python-dir]\`
- Running Python on Windows
  - `start → run`
  - In the dialog box, type `cmd` and press `[enter]` key
  - Type `python` and ensure there are no errors

Installation & Running (2)

- [Mac OS X]
  - use Homebrew
    - `brew install python3`
- Running Python on Mac OS X
  - Open the terminal by pressing `[Command + Space]` keys
  - Type `Terminal` and press `[enter]` key
  - Run `python3` and ensure there are no errors

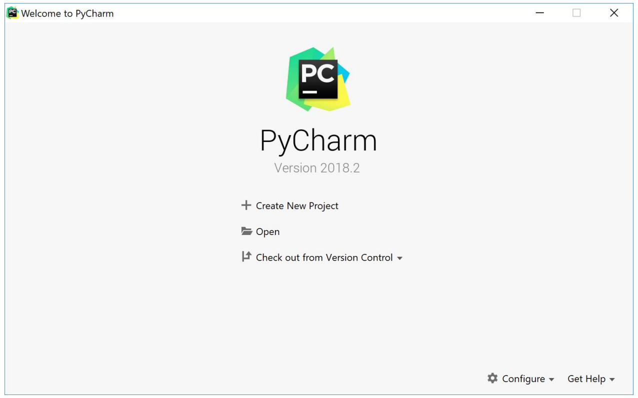
Installation & Running (3)

- [GNU/Linux]
  - use your distribution's package manager to install Python 3, e.g. on Debian & Ubuntu
    - `sudo apt-get update && sudo apt-get install python3`
- Running Python on GNU/Linux
  - Open the `Terminal` application
  - Run `python3` and ensure there are no errors

PyCharm

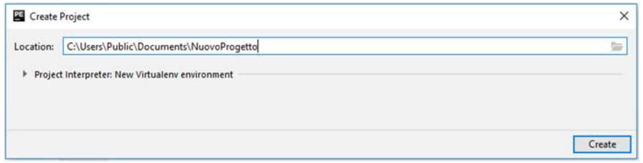
- PyCharm is a dedicated Python and Django IDE providing a wide range of essential tools for Python developers
- PyCharm permits to create a convenient environment for productive Python development
- PyCharm can be downloaded by the following link:
  - https://www.jetbrains.com/pycharm/download/

Welcome to PyCharm



Creating a project

- Creating a project in PyCharm:
  - Select `Create New Project`, or
  - `File -> New Project`
  - Define the `path` in which all the project's files will be stored
  - Click `Create`!



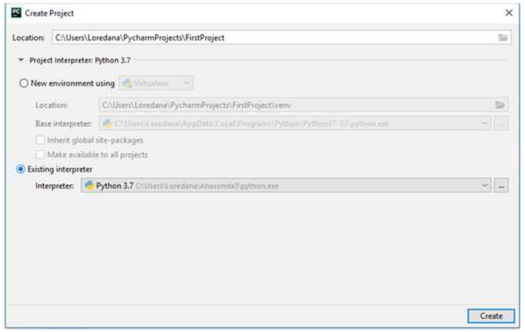
Create Project

Location: C:\Users\Public\Documents\NuovoProgetto

Project Interpreter: New Virtualenv environment

Choosing the Interpreter

- Python is a script language, which means that your code is converted to machine code by a Python interpreter



Create Project

Location: C:\Users\Loredana\PycharmProjects\FirstProject

Project Interpreter: Python 3.7

○ New environment using Virtualenv

Location: C:\Users\Loredana\PycharmProjects\FirstProject\venv

Base interpreter: C:\Users\Loredana\AppData\Local\Programs\Python\Python37-32\python.exe

□ Inherit global site-packages

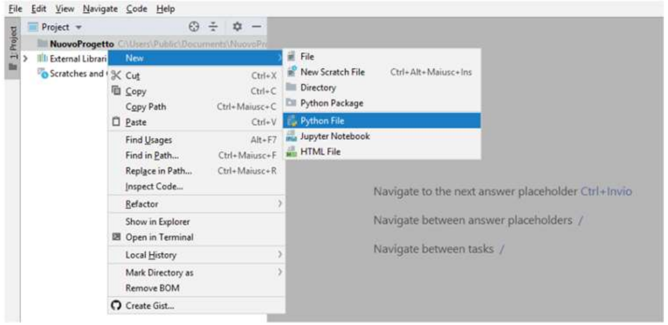
□ Make available to all projects

● Existing interpreter

Interpreter: Python 3.7 C:\Users\Loredana\Anaconda3\python.exe

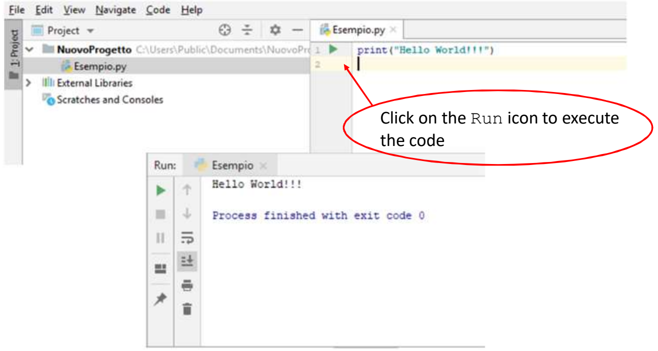
Creating a Python File

- Select the project root in the Project tool window
- Define the file name



Editing Source Code

- Python Hello World:



File Edit View Navigate Code Help
Project
NuovoProgetto C:\Users\Public\Documents\NuovoPr
Esempio.py
External Libraries
Scratches and Consoles
Esempio.py
Run: Esempio

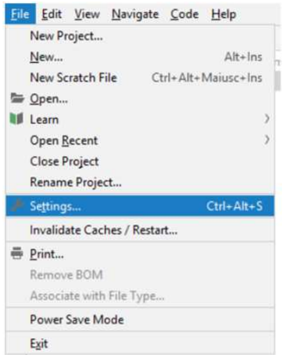
```
print("Hello World!!!")
```

Click on the `Run` icon to execute the code

```
Hello World!!!

Process finished with exit code 0
```

Installing libraries (1)

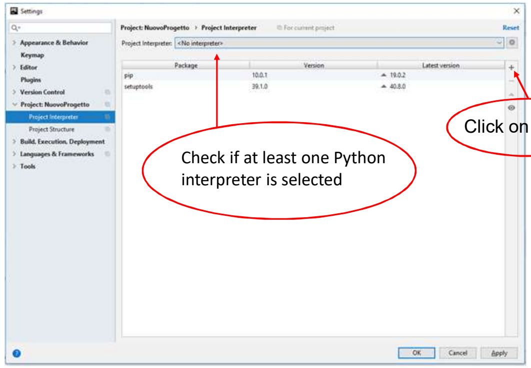
- Installing libraries in PyCharm:



Click on **`Settings`**...

Installing libraries (2)

- Installing libraries in PyCharm:



Project Interpreter: <No interpreter>

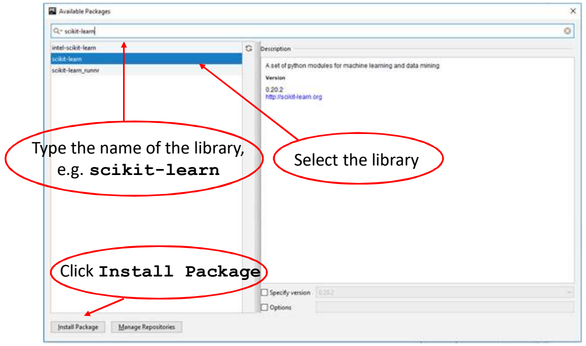
Package	Version	Latest version
pip	10.0.1	▲ 19.0.2
setuptools	39.1.0	▲ 40.8.0

Check if at least one Python interpreter is selected

Click on + icon

Installing libraries (3)

- Installing libraries in PyCharm:



scikit-learn

intel-scikit-learn
scikit-learn
scikit-learn_runnr

A set of python modules for machine learning and data mining

Version

0.20.2
http://scikit-learn.org

□ Specify version 0.20.2

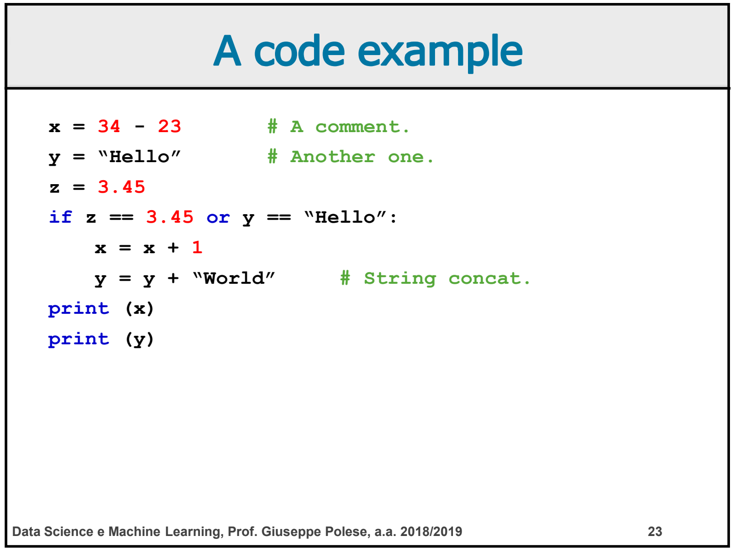
□ Options

Type the name of the library, e.g. **`scikit-learn`**

Select the library

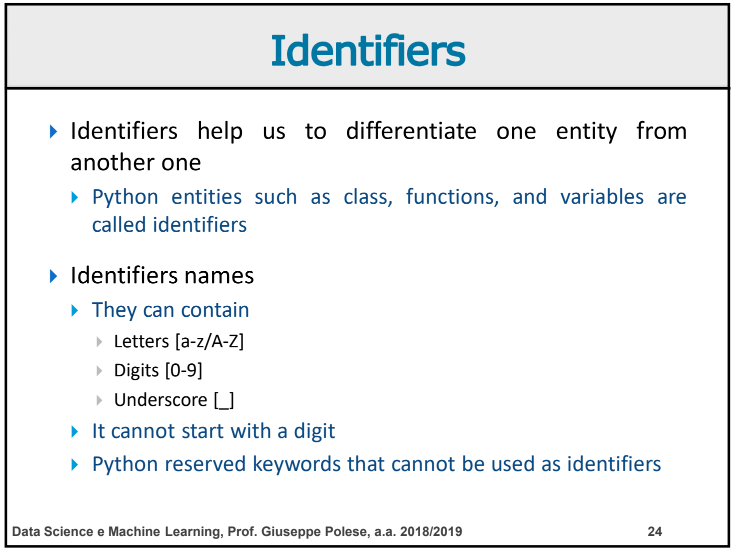
Click **`Install Package`**

The Basics



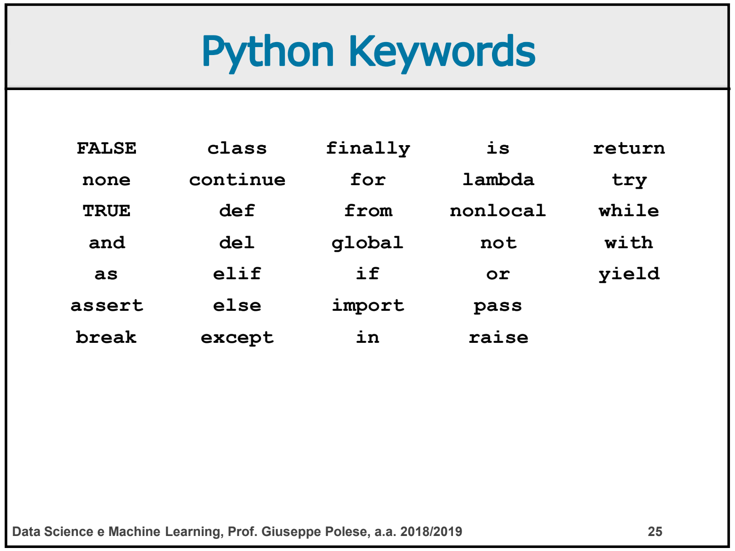
A code example

```
x = 34 - 23        # A comment.
y = “Hello”        # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “World”      # String concat.
print (x)
print (y)
```



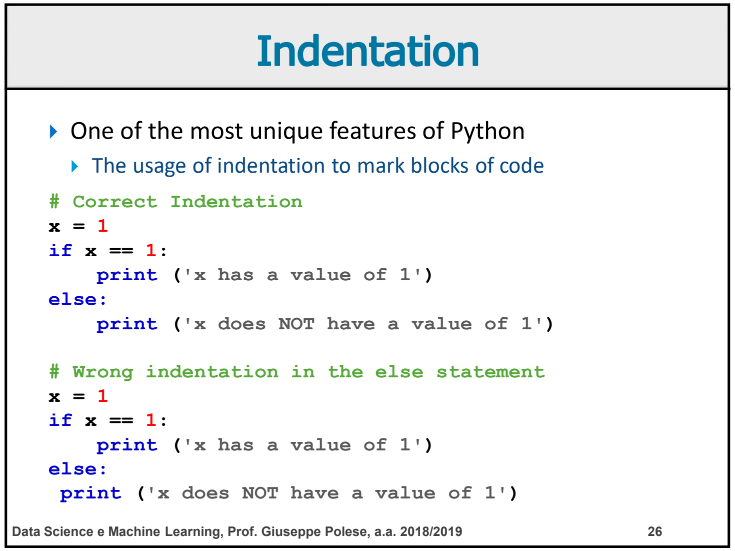
Identifiers

- Identifiers help us to differentiate one entity from another one
  - Python entities such as class, functions, and variables are called identifiers
- Identifiers names
  - They can contain
    - Letters [a-z/A-Z]
    - Digits [0-9]
    - Underscore [_]
  - It cannot start with a digit
  - Python reserved keywords that cannot be used as identifiers



Python Keywords

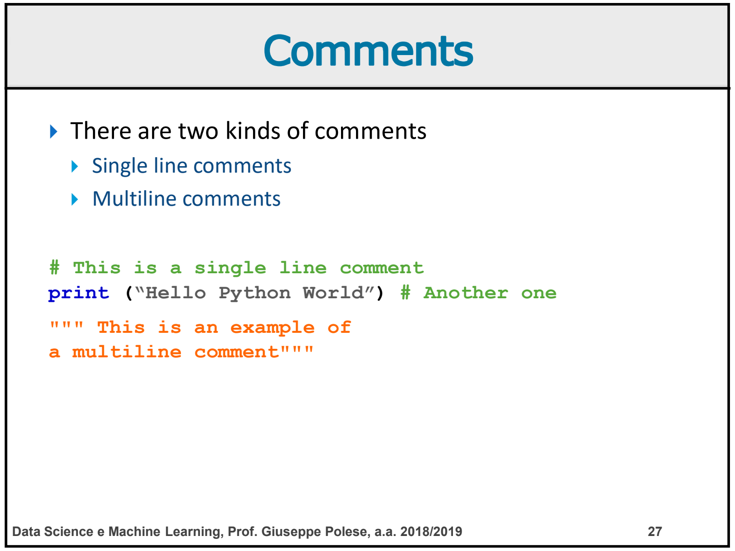
```
FALSE
none
TRUE
and
as
assert
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield
```



Indentation

- One of the most unique features of Python
  - The usage of indentation to mark blocks of code

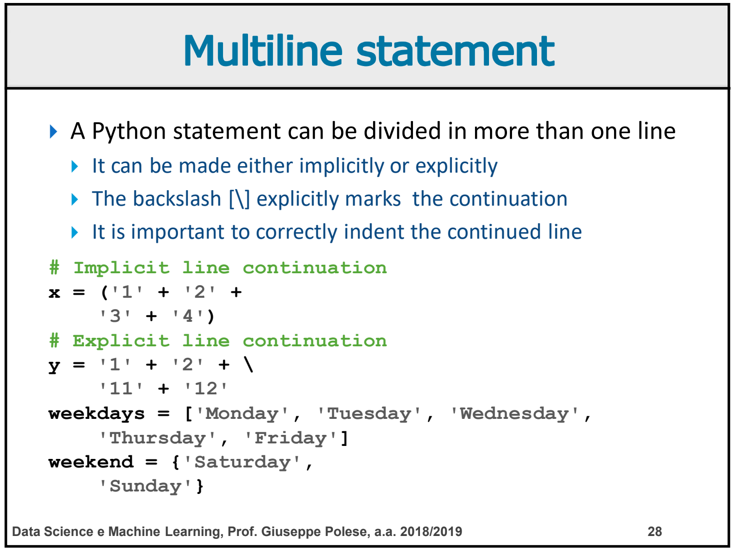
```
# Correct Indentation
x = 1
if x == 1:
    print ('x has a value of 1')
else:
    print ('x does NOT have a value of 1')

# Wrong indentation in the else statement
x = 1
if x == 1:
    print ('x has a value of 1')
else:
 print ('x does NOT have a value of 1')
```



Comments

- There are two kinds of comments
  - Single line comments
  - Multiline comments

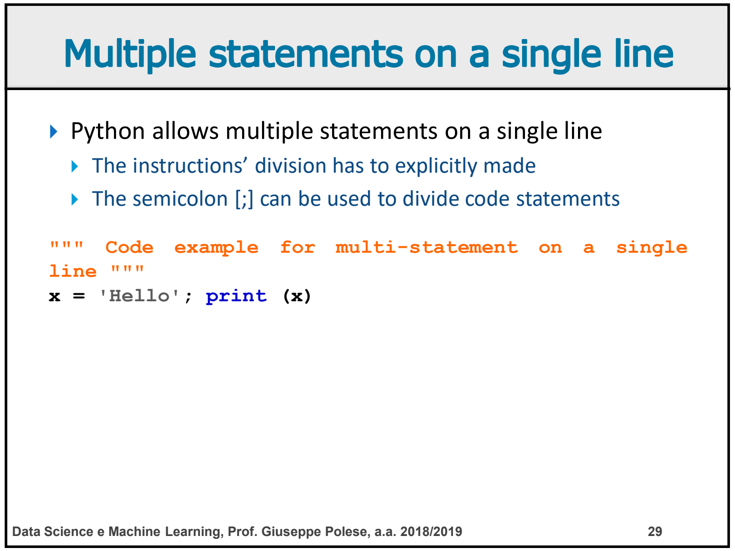
```
# This is a single line comment
print (“Hello Python World”) # Another one
""" This is an example of
a multiline comment"""
```



Multiline statement

- A Python statement can be divided in more than one line
  - It can be made either implicitly or explicitly
  - The backslash [\] explicitly marks the continuation
  - It is important to correctly indent the continued line

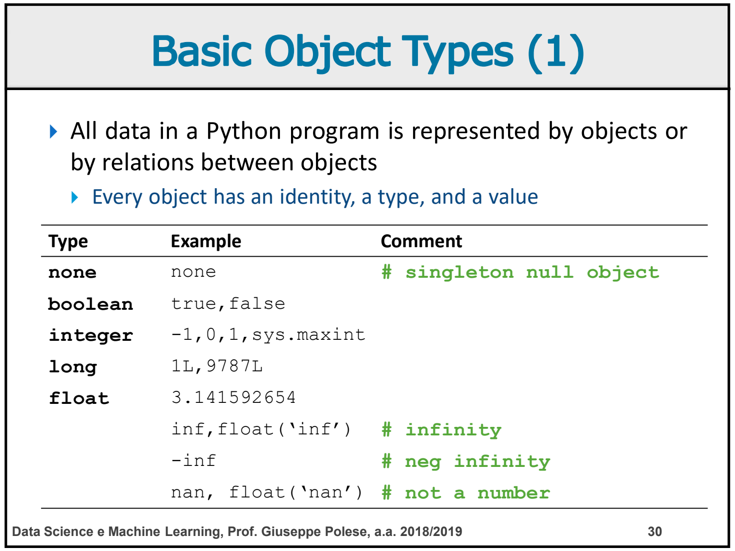
```
# Implicit line continuation
x = ('1' + '2' +
    '3' + '4')
# Explicit line continuation
y = '1' + '2' + \
    '11' + '12'
weekdays = ['Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday']
weekend = {'Saturday',
    'Sunday'}
```



Multiple statements on a single line

- Python allows multiple statements on a single line
  - The instructions’ division has to explicitly made
  - The semicolon [;] can be used to divide code statements

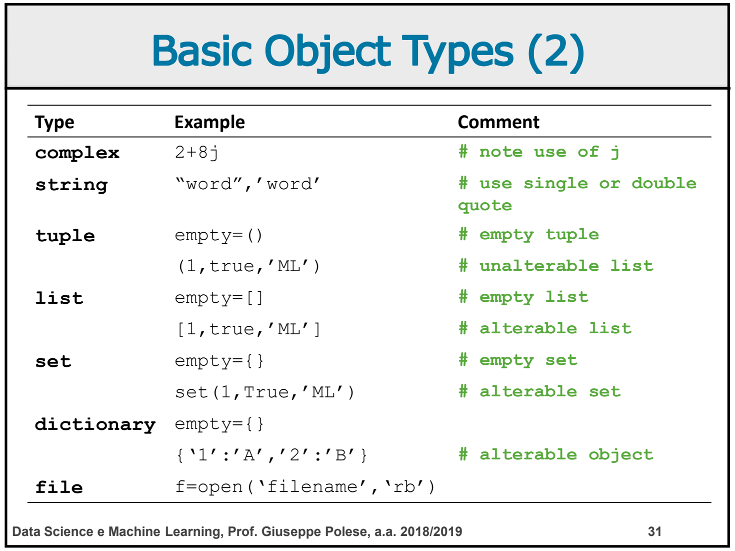
```
""" Code example for multi-statement on a single
line """
x = 'Hello'; print (x)
```



Basic Object Types (1)

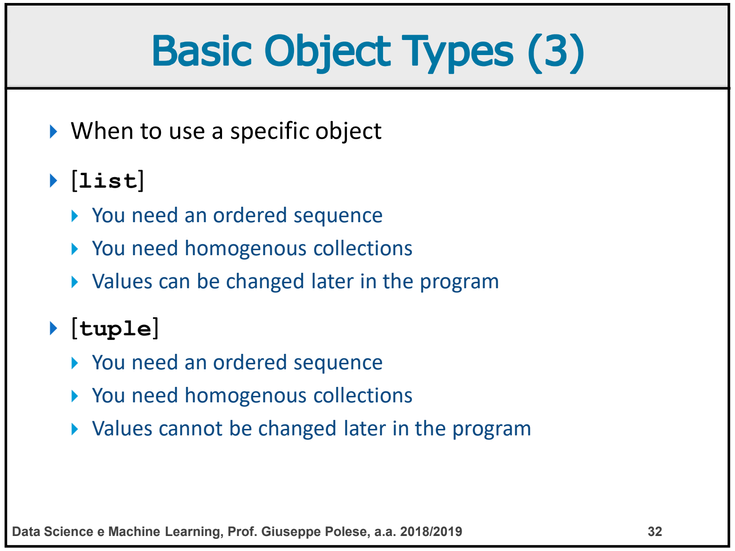
- All data in a Python program is represented by objects or by relations between objects
  - Every object has an identity, a type, and a value

Type	Example	Comment
none	none	# singleton null object
boolean	true,false	
integer	-1,0,1,sys.maxint	
long	1L,9787L	
float	3.141592654	
	inf,float(‘inf’)	# infinity
	-inf	# neg infinity
	nan, float(‘nan’)	# not a number



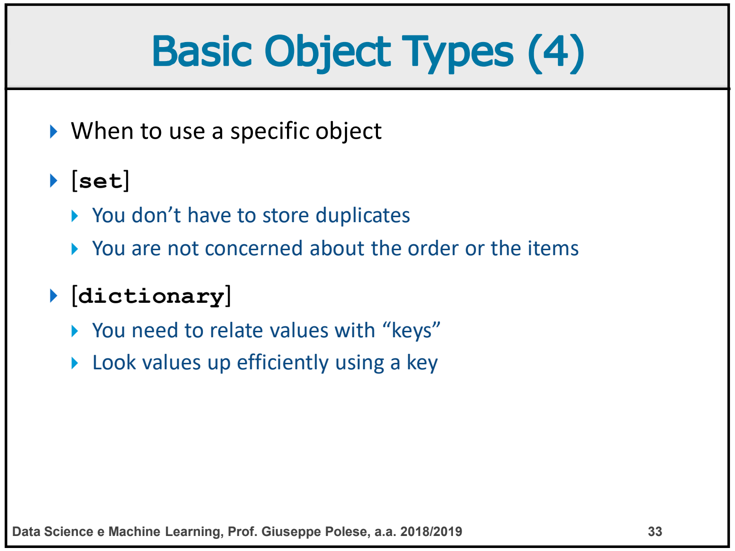
Basic Object Types (2)

Type	Example	Comment
`complex`	`2+8j`	`# note use of j`
`string`	`“word”,’word’`	`# use single or double quote`
`tuple`	`empty=()`	`# empty tuple`
	`(1,true,’ML’)`	`# unalterable list`
`list`	`empty=[]`	`# empty list`
	`[1,true,’ML’]`	`# alterable list`
`set`	`empty={}`	`# empty set`
	`set(1,True,’ML’)`	`# alterable set`
`dictionary`	`empty={}`	
	`{‘1’:’A’,’2’:’B’}`	`# alterable object`
`file`	`f=open(‘filename’,‘rb’)`	



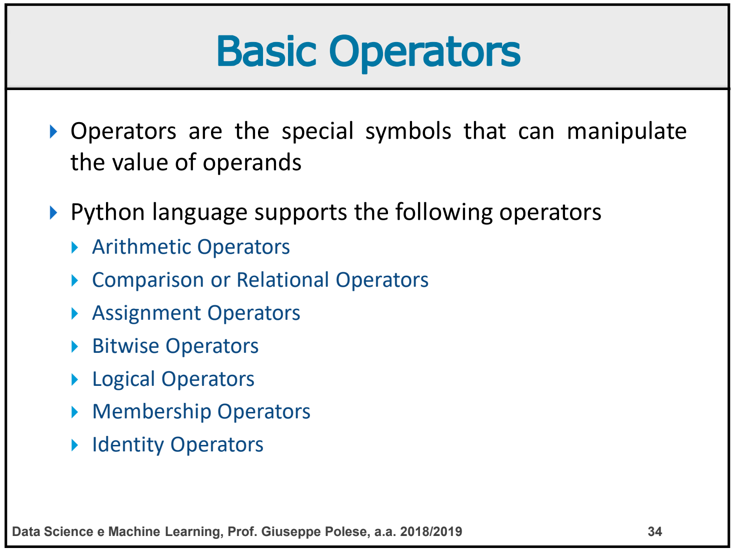
Basic Object Types (3)

- When to use a specific object
- [`list`]
  - You need an ordered sequence
  - You need homogenous collections
  - Values can be changed later in the program
- [`tuple`]
  - You need an ordered sequence
  - You need homogenous collections
  - Values cannot be changed later in the program



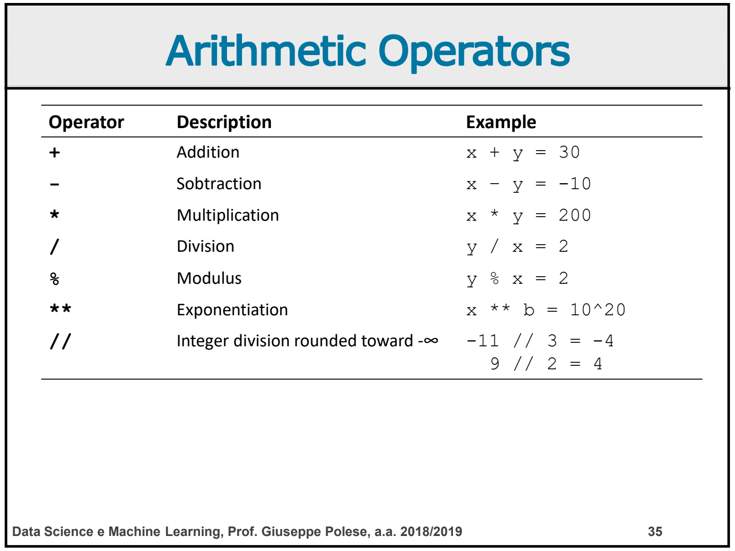
Basic Object Types (4)

- When to use a specific object
- [`set`]
  - You don’t have to store duplicates
  - You are not concerned about the order or the items
- [`dictionary`]
  - You need to relate values with “keys”
  - Look values up efficiently using a key



Basic Operators

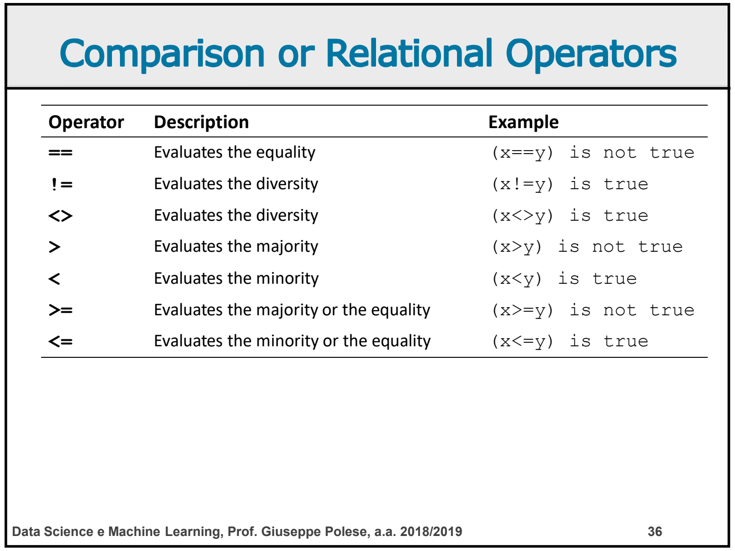
- Operators are the special symbols that can manipulate the value of operands
- Python language supports the following operators
  - Arithmetic Operators
  - Comparison or Relational Operators
  - Assignment Operators
  - Bitwise Operators
  - Logical Operators
  - Membership Operators
  - Identity Operators



Arithmetic Operators

Operator	Description	Example
+	Addition	x + y = 30
-	Sobtraction	x - y = -10
*	Multiplication	x * y = 200
/	Division	y / x = 2
%	Modulus	y % x = 2
**	Exponentiation	x ** b = 10^20
//	Integer division rounded toward -∞	-11 // 3 = -4 9 // 2 = 4

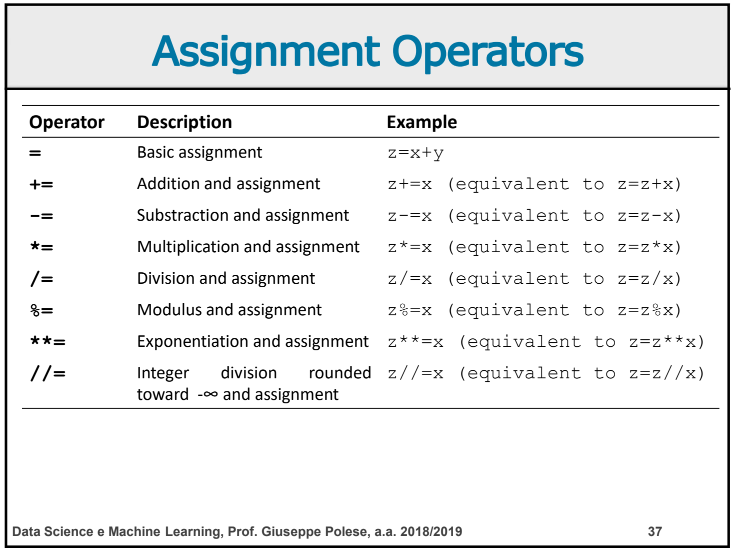
35



Comparison or Relational Operators

Operator	Description	Example
==	Evaluates the equality	(x==y) is not true
!=	Evaluates the diversity	(x!=y) is true
<>	Evaluates the diversity	(x<>y) is true
>	Evaluates the majority	(x>y) is not true
<	Evaluates the minority	(x<y) is true
>=	Evaluates the majority or the equality	(x>=y) is not true
<=	Evaluates the minority or the equality	(x<=y) is true

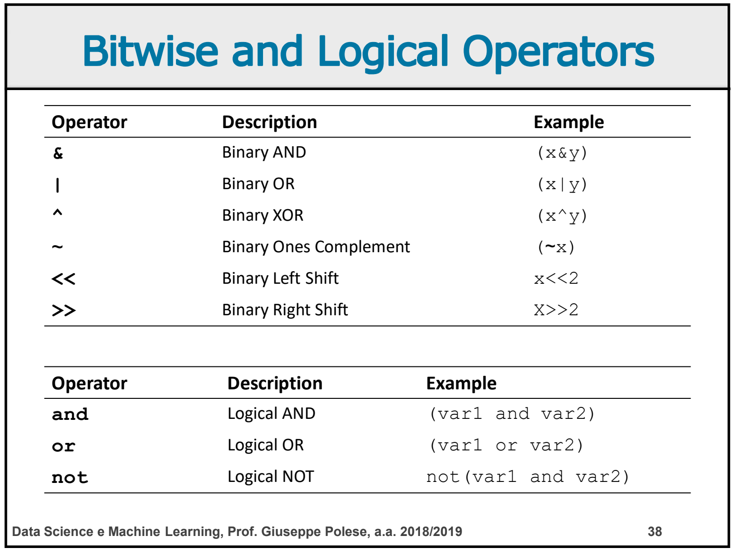
36



Assignment Operators

Operator	Description	Example
=	Basic assignment	z=x+y
+=	Addition and assignment	z+=x (equivalent to z=z+x)
-=	Substraction and assignment	z-=x (equivalent to z=z-x)
*=	Multiplication and assignment	z*=x (equivalent to z=z*x)
/=	Division and assignment	z/=x (equivalent to z=z/x)
%=	Modulus and assignment	z%=x (equivalent to z=z%x)
**=	Exponentiation and assignment	z**=x (equivalent to z=z**x)
//=	Integer division rounded toward -∞ and assignment	z//=x (equivalent to z=z//x)

37

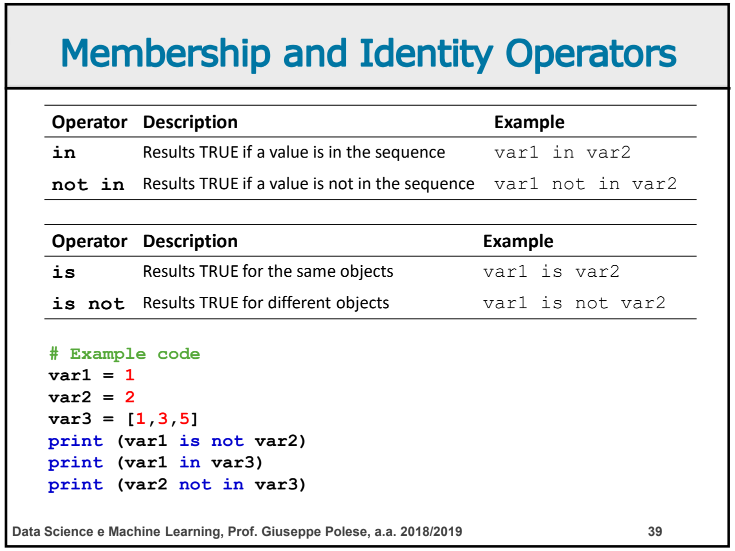


Bitwise and Logical Operators

Operator	Description	Example
&	Binary AND	(x&y)
|	Binary OR	(x|y)
^	Binary XOR	(x^y)
~	Binary Ones Complement	(~x)
<<	Binary Left Shift	x<<2
>>	Binary Right Shift	X>>2

Operator	Description	Example
and	Logical AND	(var1 and var2)
or	Logical OR	(var1 or var2)
not	Logical NOT	not(var1 and var2)

38

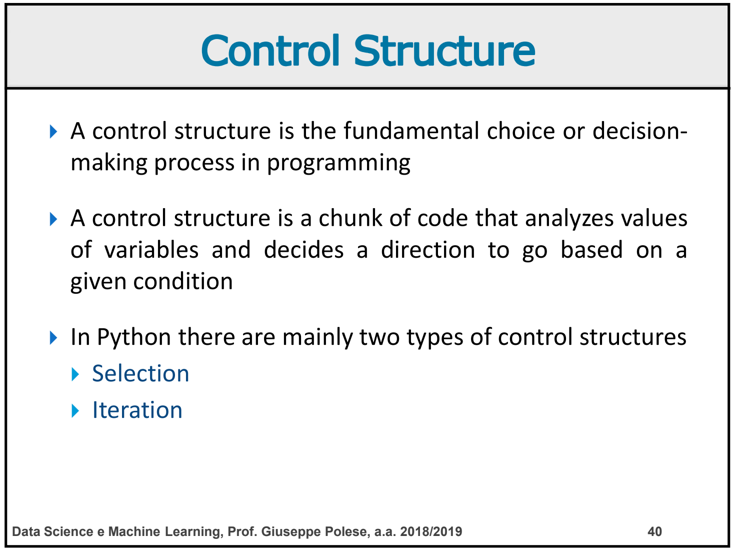


Membership and Identity Operators

Operator	Description	Example
`in`	Results TRUE if a value is in the sequence	`var1 in var2`
`not in`	Results TRUE if a value is not in the sequence	`var1 not in var2`

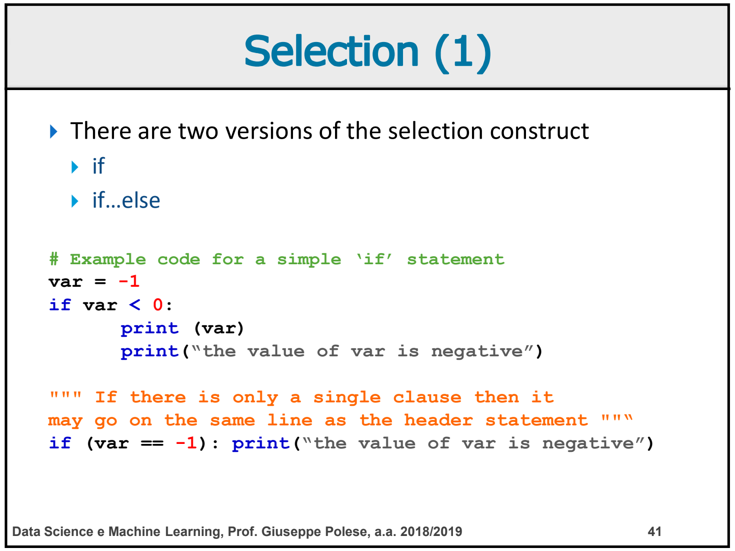
Operator	Description	Example
`is`	Results TRUE for the same objects	`var1 is var2`
`is not`	Results TRUE for different objects	`var1 is not var2`

```
# Example code
var1 = 1
var2 = 2
var3 = [1,3,5]
print (var1 is not var2)
print (var1 in var3)
print (var2 not in var3)
```



Control Structure

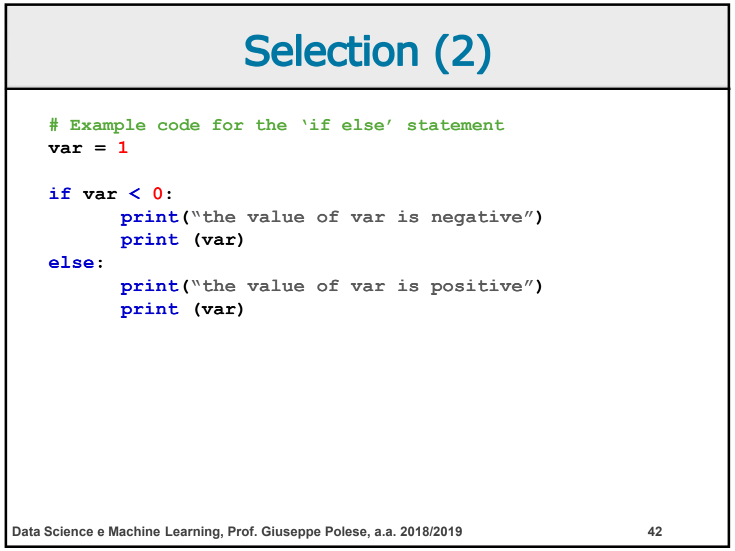
- A control structure is the fundamental choice or decision-making process in programming
- A control structure is a chunk of code that analyzes values of variables and decides a direction to go based on a given condition
- In Python there are mainly two types of control structures
  - Selection
  - Iteration



Selection (1)

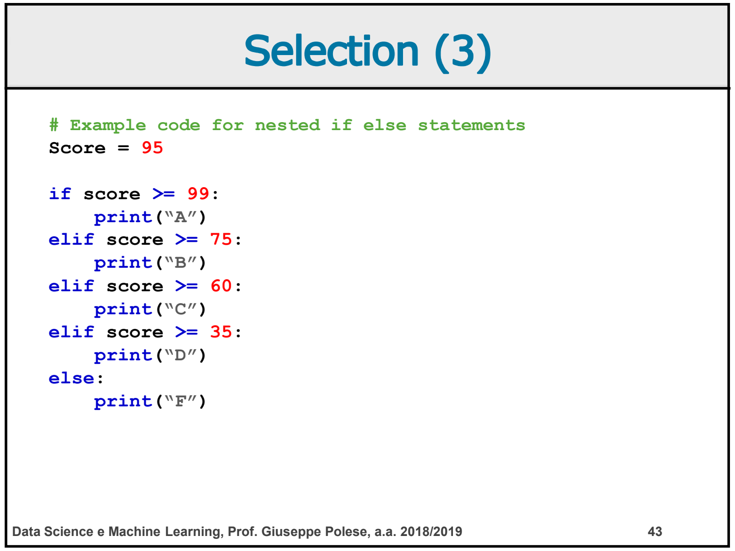
- There are two versions of the selection construct
  - if
  - if…else

```
# Example code for a simple 'if' statement
var = -1
if var < 0:
       print (var)
       print("the value of var is negative")

""" If there is only a single clause then it
may go on the same line as the header statement """
if (var == -1): print("the value of var is negative")
```



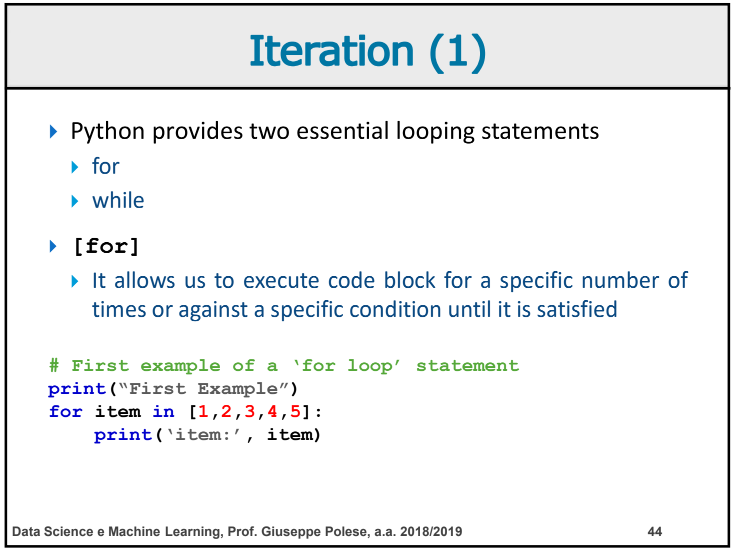
Selection (2)

```
# Example code for the 'if else' statement
var = 1

if var < 0:
       print("the value of var is negative")
       print (var)
else:
       print("the value of var is positive")
       print (var)
```



Selection (3)

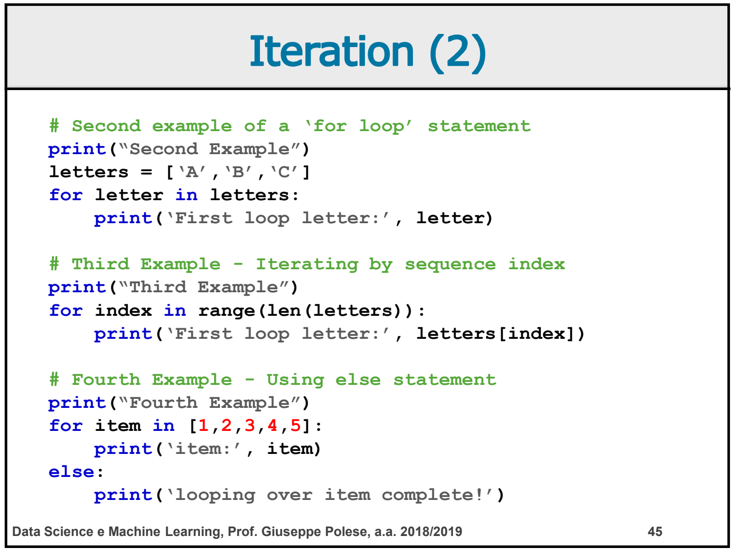
```
# Example code for nested if else statements
Score = 95

if score >= 99:
    print(“A”)
elif score >= 75:
    print(“B”)
elif score >= 60:
    print(“C”)
elif score >= 35:
    print(“D”)
else:
    print(“F”)
```



Iteration (1)

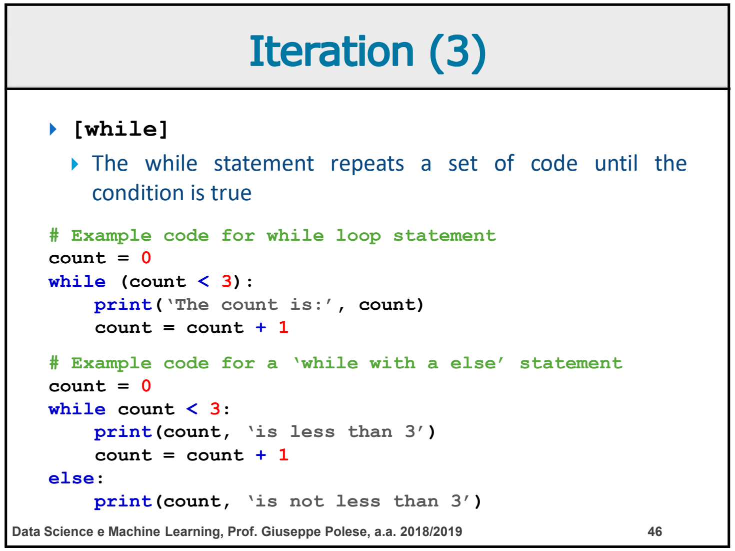
- Python provides two essential looping statements
  - for
  - while
- `[for]`
  - It allows us to execute code block for a specific number of times or against a specific condition until it is satisfied

```
# First example of a ‘for loop’ statement
print(“First Example”)
for item in [1,2,3,4,5]:
    print(‘item:’, item)
```



Iteration (2)

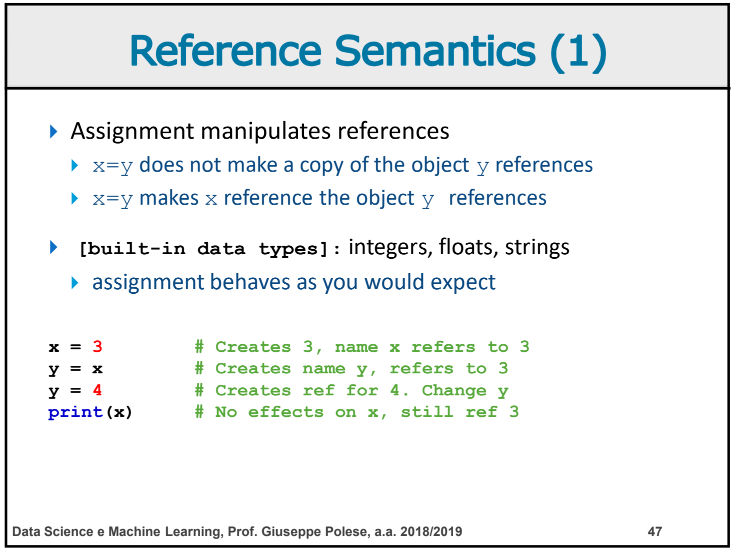
```
# Second example of a ‘for loop’ statement
print(“Second Example”)
letters = [‘A’,‘B’,‘C’]
for letter in letters:
    print(‘First loop letter:’, letter)

# Third Example - Iterating by sequence index
print(“Third Example”)
for index in range(len(letters)):
    print(‘First loop letter:’, letters[index])

# Fourth Example - Using else statement
print(“Fourth Example”)
for item in [1,2,3,4,5]:
    print(‘item:’, item)
else:
    print(‘looping over item complete!’)
```



Iteration (3)

- `[while]`
  - The while statement repeats a set of code until the condition is true

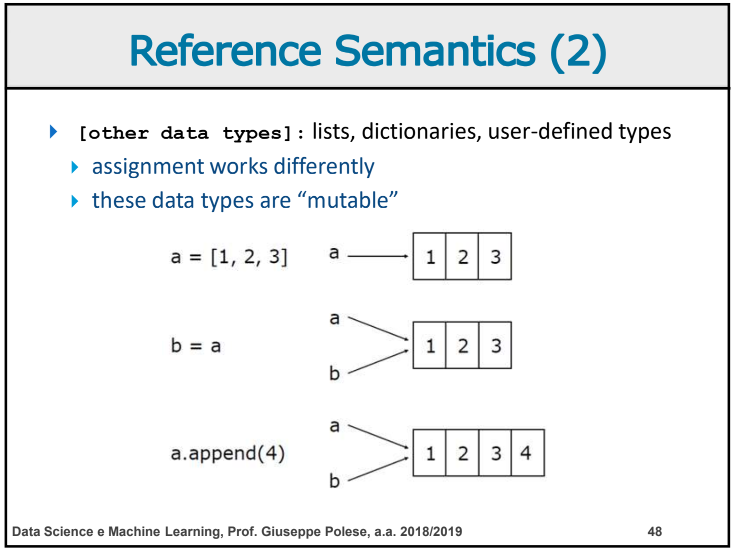
```
# Example code for while loop statement
count = 0
while (count < 3):
    print(‘The count is:’, count)
    count = count + 1

# Example code for a ‘while with a else’ statement
count = 0
while count < 3:
    print(count, ‘is less than 3’)
    count = count + 1
else:
    print(count, ‘is not less than 3’)
```



Reference Semantics (1)

- Assignment manipulates references
  - x=y does not make a copy of the object y references
  - x=y makes x reference the object y references
- **[built-in data types]**: integers, floats, strings
  - assignment behaves as you would expect

```
x = 3          # Creates 3, name x refers to 3
y = x          # Creates name y, refers to 3
y = 4          # Creates ref for 4. Change y
print(x)       # No effects on x, still ref 3
```



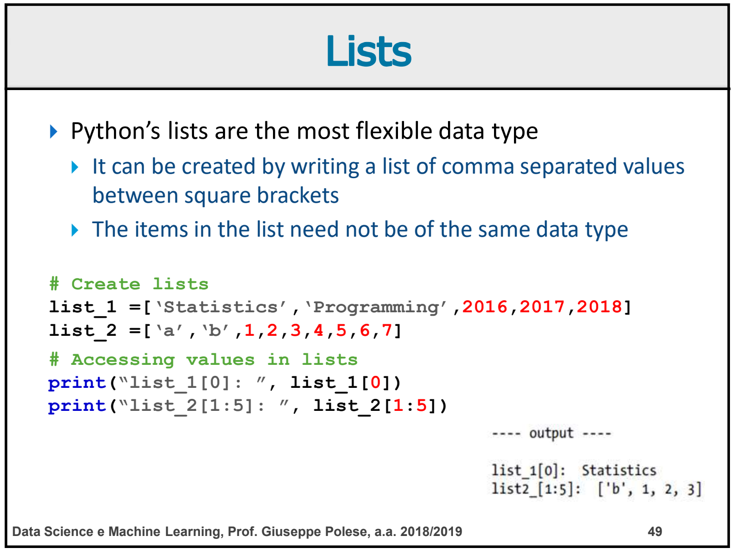
Reference Semantics (2)

- **[other data types]**: lists, dictionaries, user-defined types
  - assignment works differently
  - these data types are “mutable”

a = [1, 2, 3]

b = a

a.append(4)

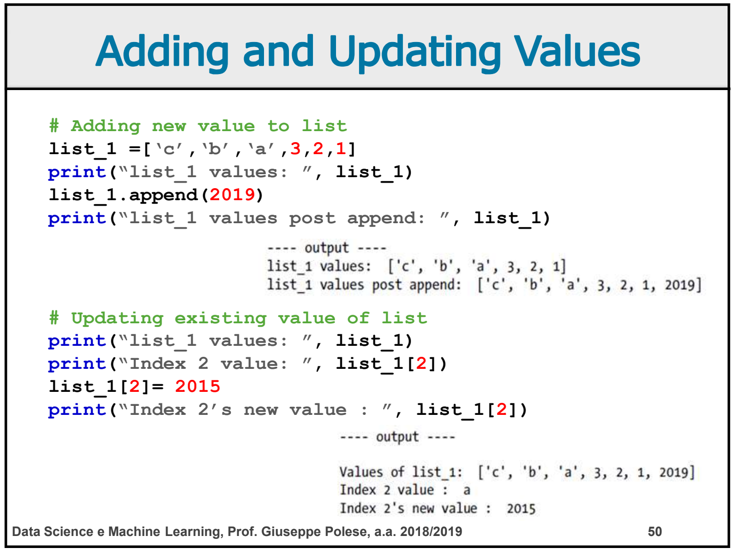


Lists

- Python’s lists are the most flexible data type
  - It can be created by writing a list of comma separated values between square brackets
  - The items in the list need not be of the same data type

```
# Create lists
list_1 =[‘Statistics’,‘Programming’,2016,2017,2018]
list_2 =[‘a’,‘b’,1,2,3,4,5,6,7]
# Accessing values in lists
print(“list_1[0]: ”, list_1[0])
print(“list_2[1:5]: ”, list_2[1:5])
```

```
---- output ----

list_1[0]:  Statistics
list2_[1:5]:  ['b', 1, 2, 3]
```



Adding and Updating Values

```
# Adding new value to list
list_1 =[‘c’,‘b’,‘a’,3,2,1]
print(“list_1 values: ”, list_1)
list_1.append(2019)
print(“list_1 values post append: ”, list_1)
```

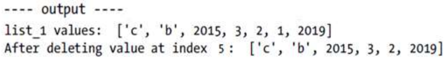
```
---- output ----
list_1 values:  ['c', 'b', 'a', 3, 2, 1]
list_1 values post append:  ['c', 'b', 'a', 3, 2, 1, 2019]
```

```
# Updating existing value of list
print(“list_1 values: ”, list_1)
print(“Index 2 value: ”, list_1[2])
list_1[2]= 2015
print(“Index 2’s new value : ”, list_1[2])
```

```
---- output ----

Values of list_1:  ['c', 'b', 'a', 3, 2, 1, 2019]
Index 2 value :  a
Index 2's new value :  2015
```

Deleting Values

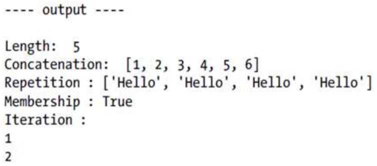
```
# Deleting list elements
list_1 =['c','b',2015,3,2,1,2019]
print("list_1 values: ", list_1)
del list_1[5]
print("After deleting value at index 5: ", list_1)
```



```
---- output ----
list_1 values:  ['c', 'b', 2015, 3, 2, 1, 2019]
After deleting value at index 5 :  ['c', 'b', 2015, 3, 2, 2019]
```

Basic Operations on Lists (1)

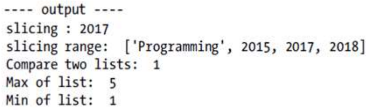
```
# Example code
list_1 =['c','b',3,2,1]
print("Length: ", len(list_1))
print("Concatenation: ", [1,2,3] + [4,5,6])
print("Repetition: ", ['Hello'] * 4)
print("Membership: ", 3 in [1,2,3])
print("Iteration: ")
for x in [1,2]: print(x)
```



```
---- output ----

Length:  5
Concatenation:  [1, 2, 3, 4, 5, 6]
Repetition : ['Hello', 'Hello', 'Hello', 'Hello']
Membership : True
Iteration :
1
2
```

Basic Operations on Lists (2)

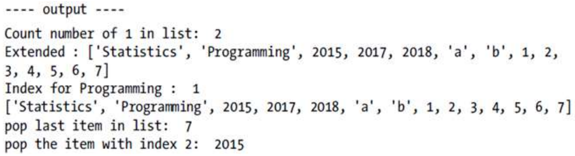
```
list_1 =['Statistics','Programming',2016,2017,2018]
# Negative sign will count from the right
print("slicing: ", list_1[-2])
"""  If  you  don't  specify  the  end  explicitly,  all
elements from the specified start index will be printed
"""
print("slicing range: ", list_1[1:])
# Comparing elements of lists
print("Compare two lists: ", cmp([1,2,3,4], [1,2,3]))
print("Max of list: ", max([1,2,3,4,5]))
print("Min of list: ", min([1,2,3,4,5]))
```



```
---- output ----
slicing : 2017
slicing range:  ['Programming', 2015, 2017, 2018]
Compare two lists:  1
Max of list:  5
Min of list:  1
```

Basic Operations on Lists (3)

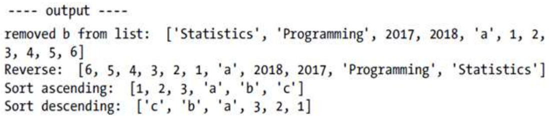
```
# Example code
print("Count number of 1: ", [1,1,2,3,4,5].count(1))
list_1 =['Statistics','Programming',2015,2017,2018]
list_2 =['a','b',1,2,3,4,5,6,7]
list_1.extend(list_2))
print("Extended: ", list_1)
print("Index for :  ", list_1.index("Programming"))
print(list_1)
print("pop last item: ", list_1.pop())
print("pop the item with index 2: ", list_1.pop(2))
```



```
---- output ----
Count number of 1 in list:  2
Extended : ['Statistics', 'Programming', 2015, 2017, 2018, 'a', 'b', 1, 2,
3, 4, 5, 6, 7]
Index for Programming :  1
['Statistics', 'Programming', 2015, 2017, 2018, 'a', 'b', 1, 2, 3, 4, 5, 6, 7]
pop last item in list:  7
pop the item with index 2:  2015
```

Basic Operations on Lists (4)

```
# Example code
list_1 =[‘Statistics’,‘Programming’,2017,
      2018,a’,‘b’,1,2,3,4,5,6]
list_1.remove(“b”)
print(“removed b from list: ”, list_1)
list_1.reverse()
print(“Reverse:  ”, list_1)
list_1 = [‘a’,‘b’,‘c’,1,2,3]
list_1.sort()
print(“Sort ascending: ”, list_1)
list_1.sort(reverse=True)
print(“Sort descending: ”, list_1)
```

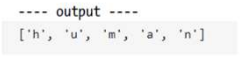


```
---- output ----
removed b from list:  ['Statistics', 'Programming', 2017, 2018, 'a', 1, 2,
3, 4, 5, 6]
Reverse:  [6, 5, 4, 3, 2, 1, 'a', 2018, 2017, 'Programming', 'Statistics']
Sort ascending:  [1, 2, 3, 'a', 'b', 'c']
Sort descending:  ['c', 'b', 'a', 3, 2, 1]
```

List Comprehension vs For Loop

- Separate the letters of the word `human` and add the letters as items of a list

```
# For Loop code
h_letters = []
for letter in ‘human’:
    h_letters.append(letter)
print(h_letters)

# List Comprehension
h_letters = [ letter for letter in ‘human’]
print(h_letters)
```

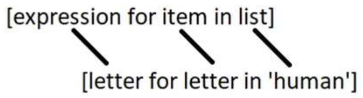


```
---- output ----
['h', 'u', 'm', 'a', 'n']
```

List Comprehension

- Syntax

```
[expression for item in list]
```



[expression for item in list]

[letter for letter in 'human']

- List Comprehension can identify when it receives a string or a tuple and works on it like a list
- Not every loop can be rewritten as list comprehension

Conditionals in List Comprehension

- List comprehensions can utilize conditional statement to modify existing list

```
# List Comprehension with conditionals
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)

# List Comprehension with nested if
num_list = [ y for y in range(100) if y%2==0 if y%5==0]
print(num_list)

# List Comprehension with if…else
obj = [‘Even’ if i%2==0 else ‘Odd’ for i in range(10)]
print(obj)
```

Outline

- The Basics
  - …
  - Data Structures
  - Functions
  - Generators
  - Class Definition
  - Final Tips

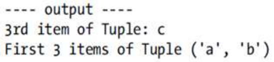
Tuple

- A Python tuple is a sequences or series of immutable Python objects very much similar to the lists
- Differences between lists and tuples
  - Unlike list, the objects of tuples cannot be changed
  - Tuples are defined by using parentheses

```
# Creating a tuple
tuple_1 = ()
tuple_2 = (1,)
tuple_3 =(‘a’,‘b’,‘c’,‘d’,1,2,3)
```

Accessing and Deleting Tuples

```
# Accessing items in tuple
print(“3rd item of Tuple: ”, tuple_3[2])
print(“First 2 items of Tuple: ”, tuple_3[0:2])
```

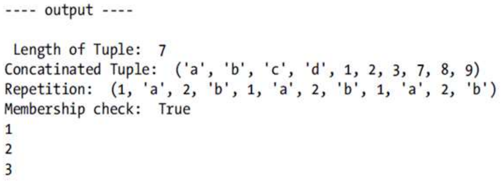


```
---- output ----
3rd item of Tuple: c
First 3 items of Tuple ('a', 'b')
```

```
# Deleting tuple
print(“Same tuple : ”, tuple_3)
del tuple_3
print(tuple_3) # Will throw an error message
```

Basic Operations on Tuples (1)

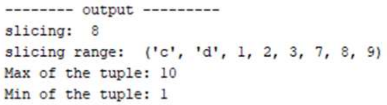
```
# Example code
tuple =(‘a’,‘b’,‘c’,‘d’,1,2,3)
print(“Length of Tuple: ”, len(tuple))
tuple_concat = tuple + (7,8,9)
print(“Concatenate tuples: ”, tuple_concat)
print(“Repetition: ”, (1,‘a’,2,‘b’) * 3)
print(“Membership check: ”, 3 in (1,2,3))
print(“Iteration: ”)
for x in [1,2,3]: print(x)
```



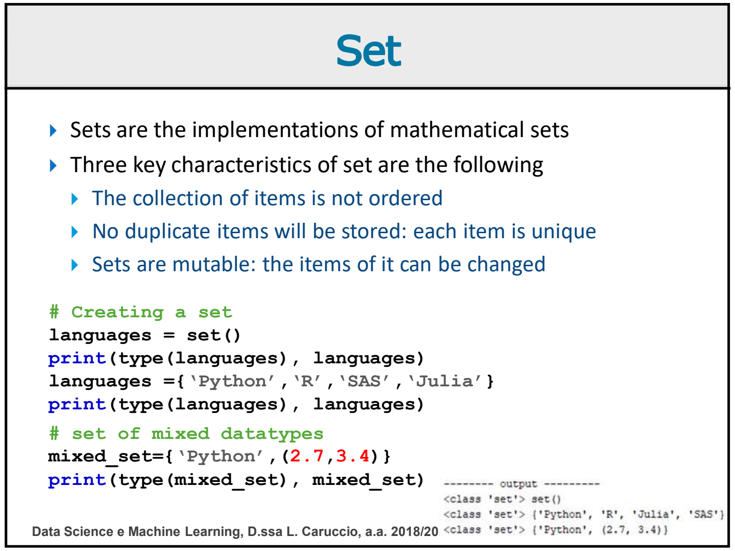
```
---- output ----

 Length of Tuple:  7
Concatinated Tuple:  ('a', 'b', 'c', 'd', 1, 2, 3, 7, 8, 9)
Repetition:  (1, 'a', 2, 'b', 1, 'a', 2, 'b', 1, 'a', 2, 'b')
Membership check:  True
1
2
3
```

Basic Operations on Tuples (2)

```
# Negative sign will retrieve item from right
print(“slicing: ”, tuple_concat[-2])
print(“slicing range: ”, tuple_concat[2:])
# Max and Min
print(“Max of the tuple:”, max((1,2,3,4,5,6,7,8,9,10)))
print(“Min of the tuple:”, min((1,2,3,4,5,6,7,8,9,10)))
```



```
-------- output ---------
slicing:  8
slicing range:  ('c', 'd', 1, 2, 3, 7, 8, 9)
Max of the tuple: 10
Min of the tuple: 1
```

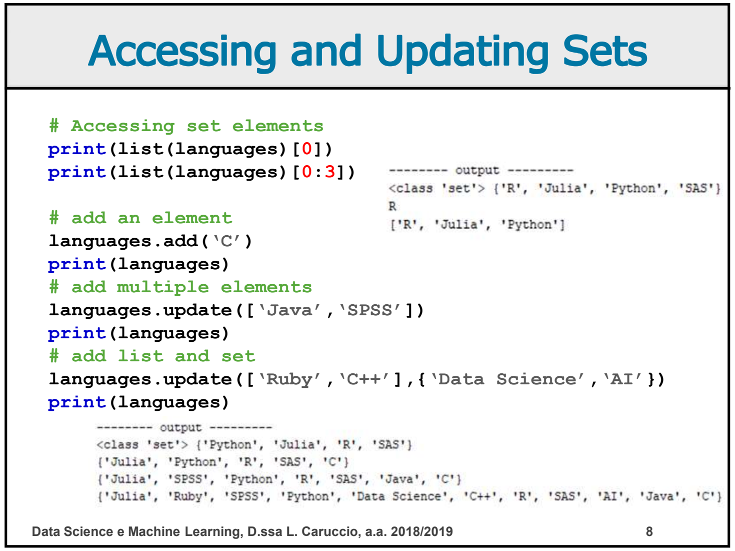


Set

- Sets are the implementations of mathematical sets
- Three key characteristics of set are the following
  - The collection of items is not ordered
  - No duplicate items will be stored: each item is unique
  - Sets are mutable: the items of it can be changed

```
# Creating a set
languages = set()
print(type(languages), languages)
languages ={‘Python’,‘R’,‘SAS’,‘Julia’}
print(type(languages), languages)
# set of mixed datatypes
mixed_set={‘Python’,(2.7,3.4)}
print(type(mixed_set), mixed_set)
```

```
-------- output ---------
<class 'set'> set()
<class 'set'> {'Python', 'R', 'Julia', 'SAS'}
<class 'set'> {'Python', (2.7, 3.4)}
```

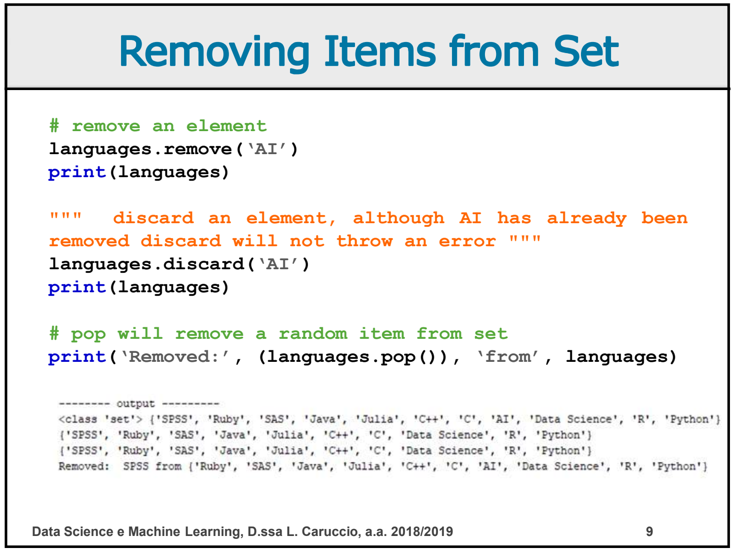


Accessing and Updating Sets

```
# Accessing set elements
print(list(languages)[0])
print(list(languages)[0:3])

# add an element
languages.add(‘C’)
print(languages)
# add multiple elements
languages.update([‘Java’,‘SPSS’])
print(languages)
# add list and set
languages.update([‘Ruby’,‘C++’],{‘Data Science’,‘AI’})
print(languages)
```

```
-------- output ---------
<class 'set'> {'R', 'Julia', 'Python', 'SAS'}
R
['R', 'Julia', 'Python']
```

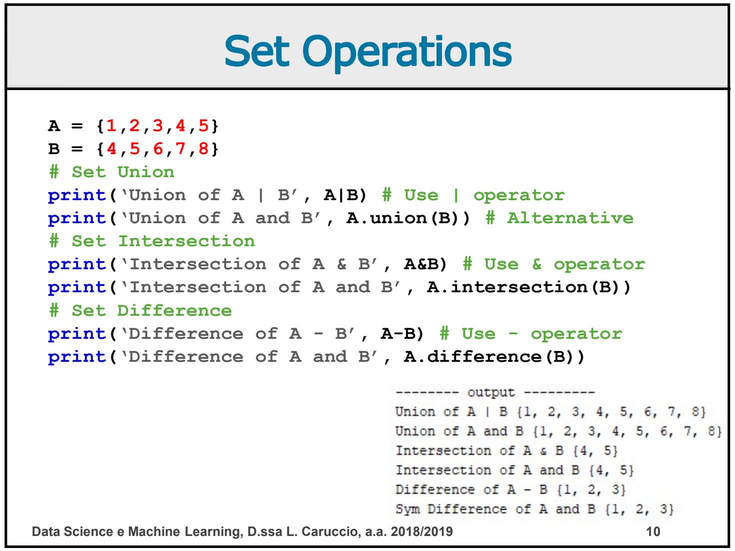
```
-------- output ---------
<class 'set'> {'Python', 'Julia', 'R', 'SAS'}
{'Julia', 'Python', 'R', 'SAS', 'C'}
{'Julia', 'SPSS', 'Python', 'R', 'SAS', 'Java', 'C'}
{'Julia', 'Ruby', 'SPSS', 'Python', 'Data Science', 'C++', 'R', 'SAS', 'AI', 'Java', 'C'}
```



Removing Items from Set

```
# remove an element
languages.remove(‘AI’)
print(languages)

"""  discard an element, although AI has already been
removed discard will not throw an error """
languages.discard(‘AI’)
print(languages)

# pop will remove a random item from set
print(‘Removed:’, (languages.pop()), ‘from’, languages)
```

```
-------- output ---------
<class 'set'> {'SPSS', 'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'AI', 'Data Science', 'R', 'Python'}
{'SPSS', 'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'Data Science', 'R', 'Python'}
{'SPSS', 'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'Data Science', 'R', 'Python'}
Removed:  SPSS from {'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'AI', 'Data Science', 'R', 'Python'}
```



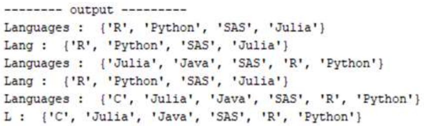
Set Operations

```
A = {1,2,3,4,5}
B = {4,5,6,7,8}
# Set Union
print(‘Union of A | B’, A|B) # Use | operator
print(‘Union of A and B’, A.union(B)) # Alternative
# Set Intersection
print(‘Intersection of A & B’, A&B) # Use & operator
print(‘Intersection of A and B’, A.intersection(B))
# Set Difference
print(‘Difference of A - B’, A-B) # Use - operator
print(‘Difference of A and B’, A.difference(B))
```

```
-------- output ---------
Union of A | B {1, 2, 3, 4, 5, 6, 7, 8}
Union of A and B {1, 2, 3, 4, 5, 6, 7, 8}
Intersection of A & B {4, 5}
Intersection of A and B {4, 5}
Difference of A - B {1, 2, 3}
Sym Difference of A and B {1, 2, 3}
```

Basic Operations on Sets (1)

```
languages ={‘Python’,‘R’,‘SAS’,‘Julia’}
# Return a shallow copy of a set
lang = languages.copy()
print(“Languages : ”, languages)
print(“Lang : ”, lang)
# Add an element in languages
languages.add(‘Java’)
print(“Languages : ”, languages)
print(“Lang : ”, lang)
l=languages
# Add an element in languages
l.add(‘C’)
print(“Languages : ”, languages)
print(“L : ”, l)
```

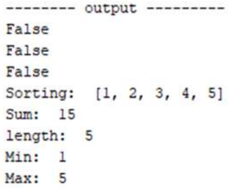


```
-------- output ---------
Languages :  {'R', 'Python', 'SAS', 'Julia'}
Lang :  {'R', 'Python', 'SAS', 'Julia'}
Languages :  {'Julia', 'Java', 'SAS', 'R', 'Python'}
Lang :  {'R', 'Python', 'SAS', 'Julia'}
Languages :  {'C', 'Julia', 'Java', 'SAS', 'R', 'Python'}
L :  {'C', 'Julia', 'Java', 'SAS', 'R', 'Python'}
```

11

Basic Operations on Sets (2)

```
A = {2,1,3,4,5}
B = {4,5,6,7,8}
#Binary operations
print(A.isdisjoint(B)) #True for non intersecting sets
print(A.issubset(B))
print(A.issuperset(B))
#Unary operations
print(“Sorting: ”, sorted(A)) #Return a new sorted list
print(“Sum: ”, sum(A)) #Return the sum of all items
print(“length: ”, len(A))
print(“Min: ”, min(A))
print(“Max: ”, max(A))
```



```
-------- output ---------
False
False
False
Sorting:  [1, 2, 3, 4, 5]
Sum:  15
length:  5
Min:  1
Max:  5
```

12

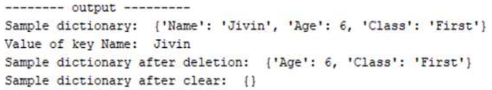
Dictionary

- The Python dictionary will have a key and value pair for each item that is part of it
- The key characteristic of dictionary are the following
  - The key and value should be enclosed in curly braces
  - Each key and value is separated using a colon [:]
  - Each item is separated by commas [,]
  - Keys are unique within a specific dictionary and must be immutable data types: strings, numbers, tuples
  - Values can take duplicate data of any type

13

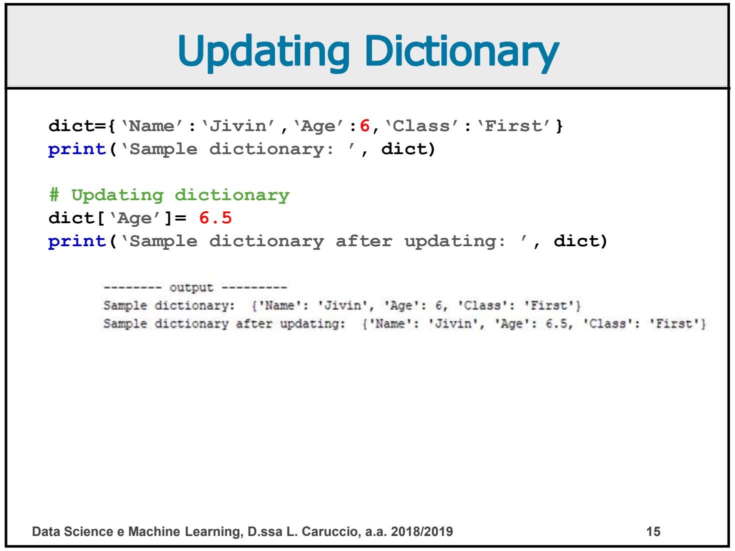
Creating and Deleting Dictionary

```
# Creating dictionary
dict={‘Name’:‘Jivin’,‘Age’:6,‘Class’:‘First’}
print(‘Sample dictionary: ’, dict)

# Accessing items in dictionary
print(‘Value of key Name: ’, dict[‘Name’])

# Example for deleting dictionary
del dict[‘Name’] # Delete specific item
print(‘Sample dictionary after deletion: ’, dict)
dict.clear() # Delete all contents
print(‘Sample dictionary after clear: ’, dict)
del dict # Delete the dictionary
```



```
-------- output ---------
Sample dictionary:  {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
Value of key Name:  Jivin
Sample dictionary after deletion:  {'Age': 6, 'Class': 'First'}
Sample dictionary after clear:  {}
```

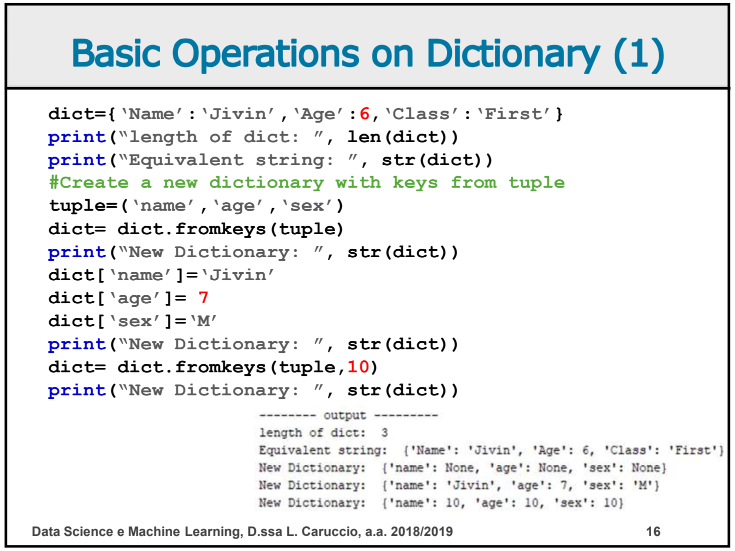
14



Updating Dictionary

```
dict={‘Name’:‘Jivin’,‘Age’:6,‘Class’:‘First’}
print(‘Sample dictionary: ’, dict)

# Updating dictionary
dict[‘Age’]= 6.5
print(‘Sample dictionary after updating: ’, dict)
```

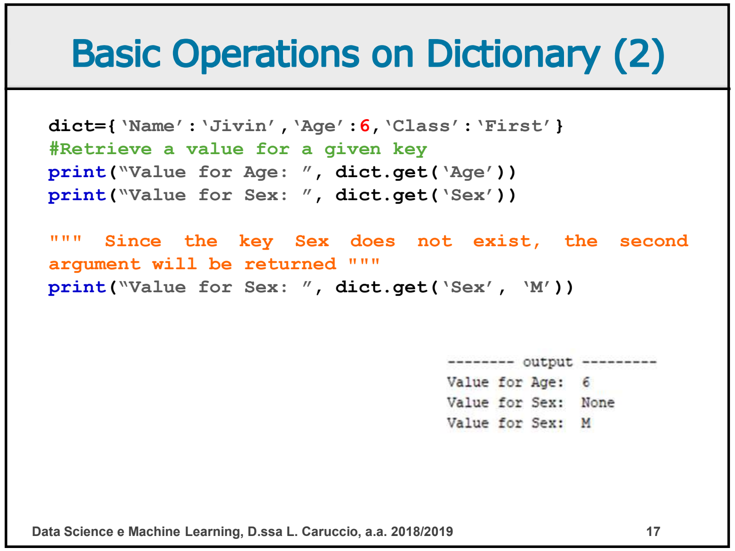
```
-------- output ---------
Sample dictionary:  {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
Sample dictionary after updating:  {'Name': 'Jivin', 'Age': 6.5, 'Class': 'First'}
```



Basic Operations on Dictionary (1)

```
dict={‘Name’:‘Jivin’,‘Age’:6,‘Class’:‘First’}
print(“length of dict: ”, len(dict))
print(“Equivalent string: ”, str(dict))
#Create a new dictionary with keys from tuple
tuple=(‘name’,‘age’,‘sex’)
dict= dict.fromkeys(tuple)
print(“New Dictionary: ”, str(dict))
dict[‘name’]=‘Jivin’
dict[‘age’]= 7
dict[‘sex’]=‘M’
print(“New Dictionary: ”, str(dict))
dict= dict.fromkeys(tuple,10)
print(“New Dictionary: ”, str(dict))
```

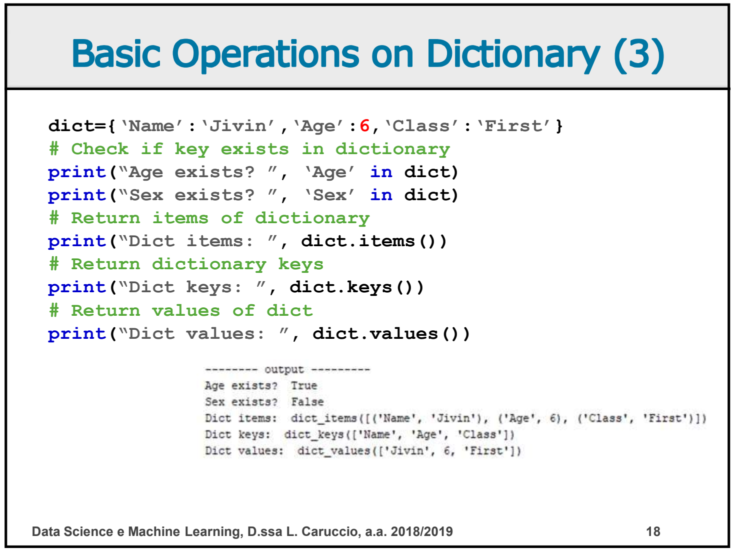
```
-------- output ---------
length of dict:  3
Equivalent string:  {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
New Dictionary:  {'name': None, 'age': None, 'sex': None}
New Dictionary:  {'name': 'Jivin', 'age': 7, 'sex': 'M'}
New Dictionary:  {'name': 10, 'age': 10, 'sex': 10}
```



Basic Operations on Dictionary (2)

```
dict={‘Name’:‘Jivin’,‘Age’:6,‘Class’:‘First’}
#Retrieve a value for a given key
print(“Value for Age: ”, dict.get(‘Age’))
print(“Value for Sex: ”, dict.get(‘Sex’))

""" Since the key Sex does not exist, the second
argument will be returned """
print(“Value for Sex: ”, dict.get(‘Sex’, ‘M’))
```

```
-------- output ---------
Value for Age:  6
Value for Sex:  None
Value for Sex:  M
```



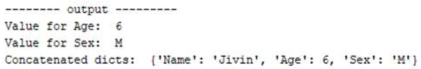
Basic Operations on Dictionary (3)

```
dict={‘Name’:‘Jivin’,‘Age’:6,‘Class’:‘First’}
# Check if key exists in dictionary
print(“Age exists? ”, ‘Age’ in dict)
print(“Sex exists? ”, ‘Sex’ in dict)
# Return items of dictionary
print(“Dict items: ”, dict.items())
# Return dictionary keys
print(“Dict keys: ”, dict.keys())
# Return values of dict
print(“Dict values: ”, dict.values())
```

```
-------- output ---------
Age exists?  True
Sex exists?  False
Dict items:  dict_items([('Name', 'Jivin'), ('Age', 6), ('Class', 'First')])
Dict keys:  dict_keys(['Name', 'Age', 'Class'])
Dict values:  dict_values(['Jivin', 6, 'First'])
```

Basic Operations on Dictionary (4)

```
dict={'Name':'Jivin','Age':6,'Class':'First'}

""" if key does not exists, then the arguments will be
added to dict and returned"""
print("Value for Age: ", dict.setdefault('Age', None))
print("Value for Sex: ", dict.setdefault('Sex', 'M'))

# Concatenate dictionaries
dict={'Name':'Jivin','Age':6}
dict2={'Sex':'M'}
dict.update(dict2)
print("Concatenated dicts: ", dict)
```



```
-------- output ---------
Value for Age:  6
Value for Sex:  M
Concatenated dicts:  {'Name': 'Jivin', 'Age': 6, 'Sex': 'M'}
```

Function (1)

- A user-defined function is a block of related code statements that are organized to achieve a single related action
- The key objective of the user-defined functions concept is to encourage modularity and enable reusability of code
- The set of rules to be followed to define a function in Python
  - The keyword `def` denotes the beginning of a function block, which will be followed by the name of the function, and open/close parentheses
  - The colon [:] has to be put to indicate the end of the function header

Function (2)

- Syntax

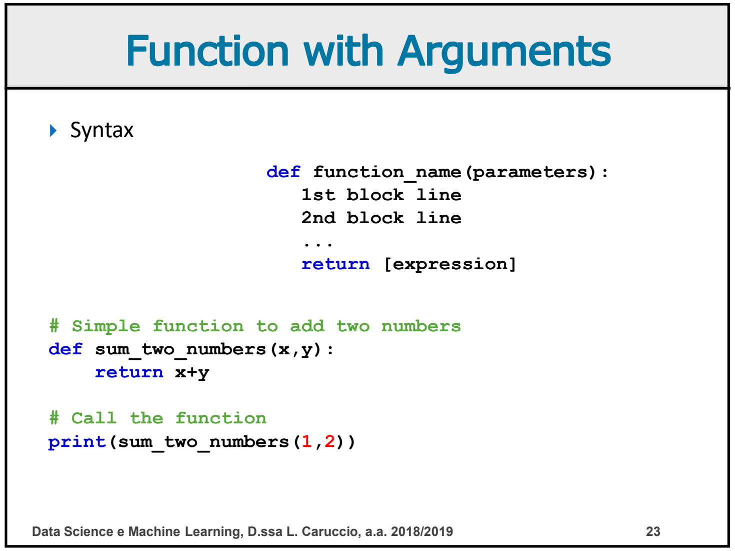
```
def function_name():
   1st block line
   2nd block line
   ...
```

- The set of rules to be followed to define a function in Python
  - Functions can accept arguments or parameters
    - Any such input should be placed within the parentheses in the header of the parameter
  - The main code statements are to be put below the function header and should be indented
    - To indicate that the code is part of the same function

Function (3)

- The set of rules to be followed to define a function in Python
  - Functions can return an expression to the caller
    - If `return` method is not used at the end of the function, it will act as a sub-procedure

```
# Simple function
def some_function():
    print("Hello World!")

# Call the function
some_function()
```

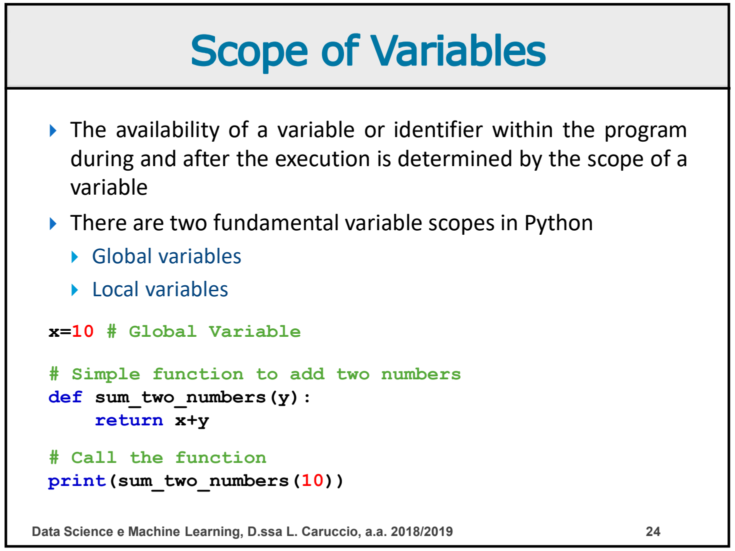


Function with Arguments

- Syntax

```
def function_name(parameters):
    1st block line
    2nd block line
    ...
    return [expression]
```

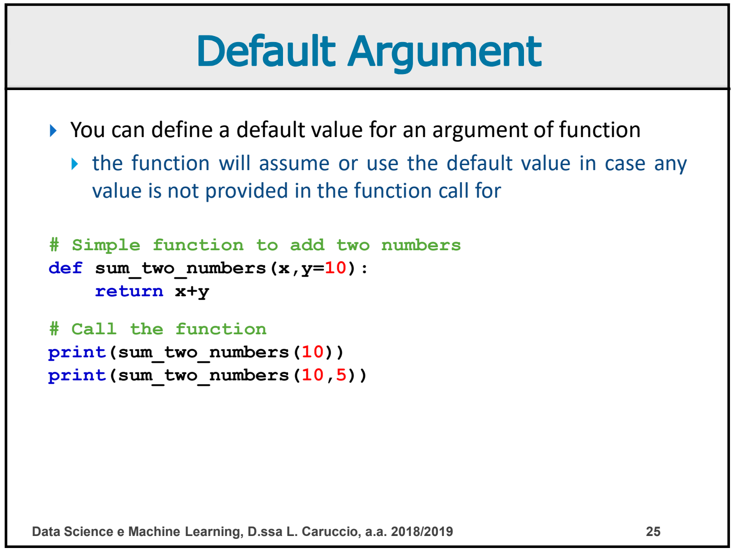
```
# Simple function to add two numbers
def sum_two_numbers(x,y):
    return x+y

# Call the function
print(sum_two_numbers(1,2))
```



Scope of Variables

- The availability of a variable or identifier within the program during and after the execution is determined by the scope of a variable
- There are two fundamental variable scopes in Python
  - Global variables
  - Local variables

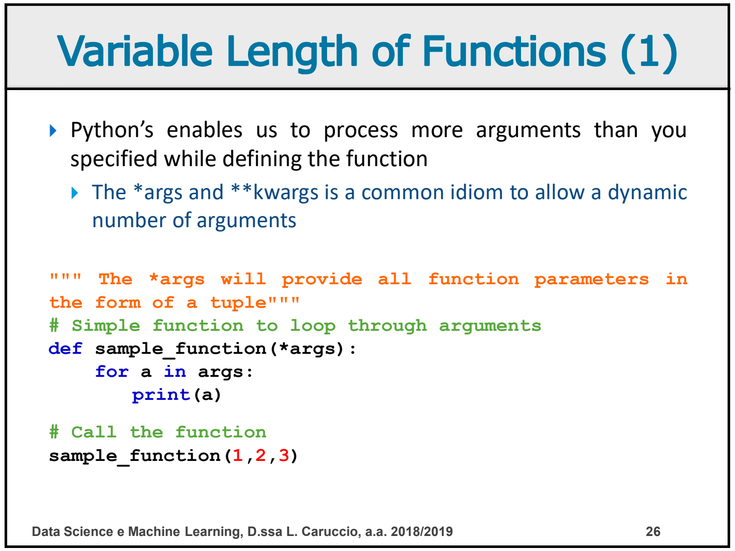
```
x=10 # Global Variable

# Simple function to add two numbers
def sum_two_numbers(y):
    return x+y

# Call the function
print(sum_two_numbers(10))
```



Default Argument

- You can define a default value for an argument of function
  - the function will assume or use the default value in case any value is not provided in the function call for

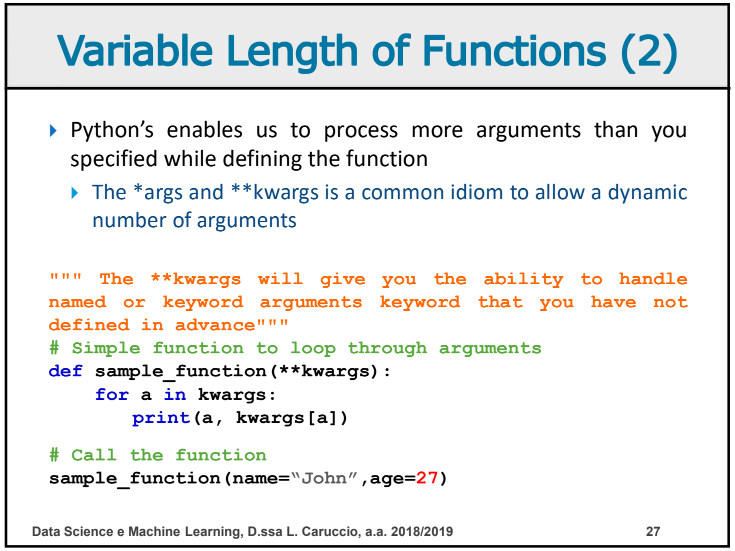
```
# Simple function to add two numbers
def sum_two_numbers(x,y=10):
    return x+y

# Call the function
print(sum_two_numbers(10))
print(sum_two_numbers(10,5))
```



Variable Length of Functions (1)

- Python’s enables us to process more arguments than you specified while defining the function
  - The *args and **kwargs is a common idiom to allow a dynamic number of arguments

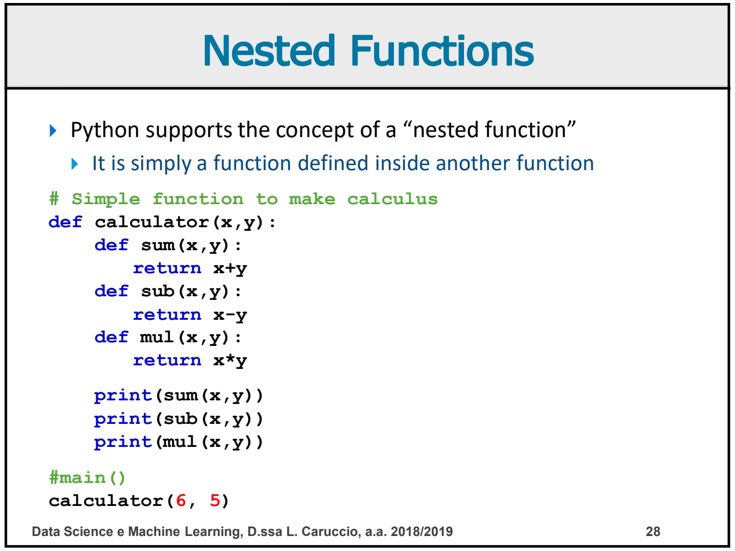
```
""" The *args will provide all function parameters in
the form of a tuple"""
# Simple function to loop through arguments
def sample_function(*args):
    for a in args:
        print(a)

# Call the function
sample_function(1,2,3)
```



Variable Length of Functions (2)

- Python’s enables us to process more arguments than you specified while defining the function
  - The *args and **kwargs is a common idiom to allow a dynamic number of arguments

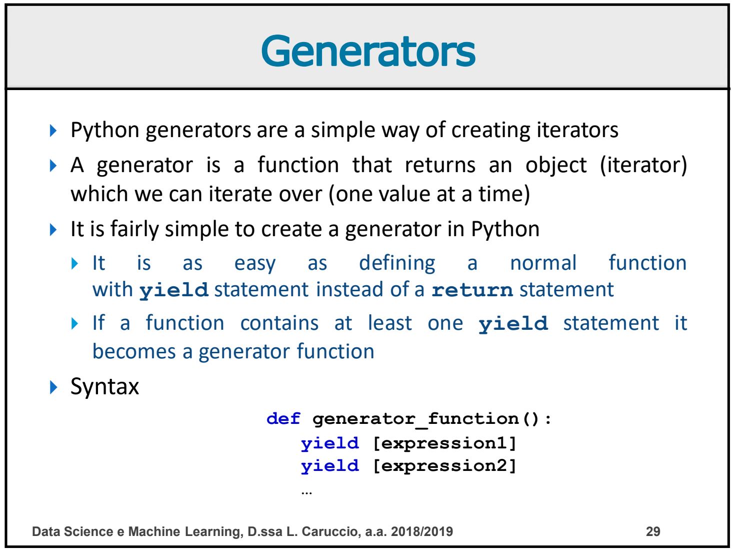
```
""" The **kwargs will give you the ability to handle
named or keyword arguments keyword that you have not
defined in advance"""
# Simple function to loop through arguments
def sample_function(**kwargs):
    for a in kwargs:
        print(a, kwargs[a])

# Call the function
sample_function(name="John",age=27)
```



Nested Functions

- Python supports the concept of a “nested function”
  - It is simply a function defined inside another function

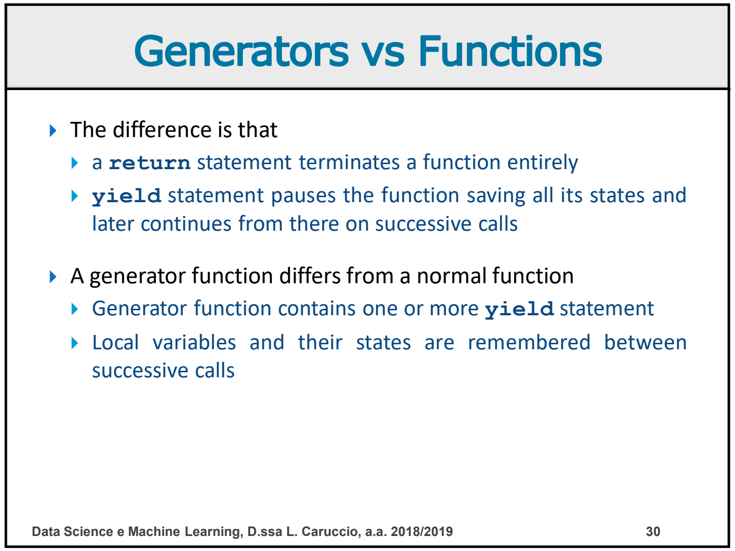
```
# Simple function to make calculus
def calculator(x,y):
    def sum(x,y):
        return x+y
    def sub(x,y):
        return x-y
    def mul(x,y):
        return x*y

    print(sum(x,y))
    print(sub(x,y))
    print(mul(x,y))

#main()
calculator(6, 5)
```



Generators

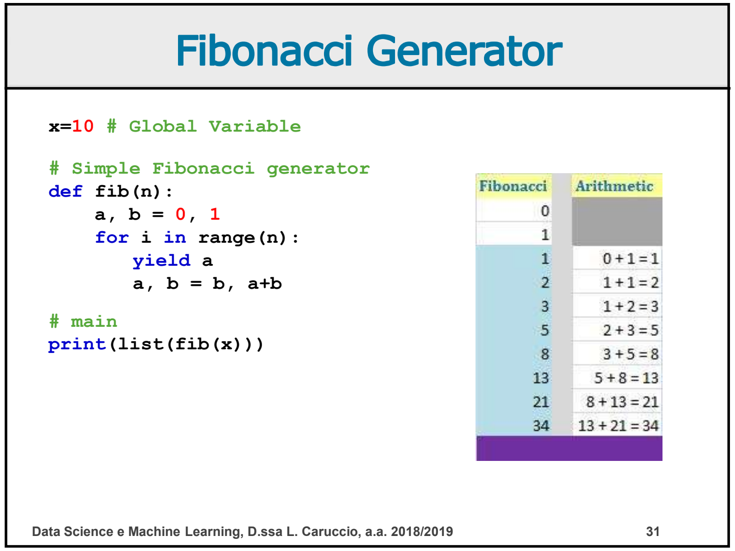
- Python generators are a simple way of creating iterators
- A generator is a function that returns an object (iterator) which we can iterate over (one value at a time)
- It is fairly simple to create a generator in Python
  - It is as easy as defining a normal function with `yield` statement instead of a `return` statement
  - If a function contains at least one `yield` statement it becomes a generator function
- Syntax

```
def generator_function():
    yield [expression1]
    yield [expression2]
    …
```



Generators vs Functions

- The difference is that
  - a `return` statement terminates a function entirely
  - `yield` statement pauses the function saving all its states and later continues from there on successive calls
- A generator function differs from a normal function
  - Generator function contains one or more `yield` statement
  - Local variables and their states are remembered between successive calls

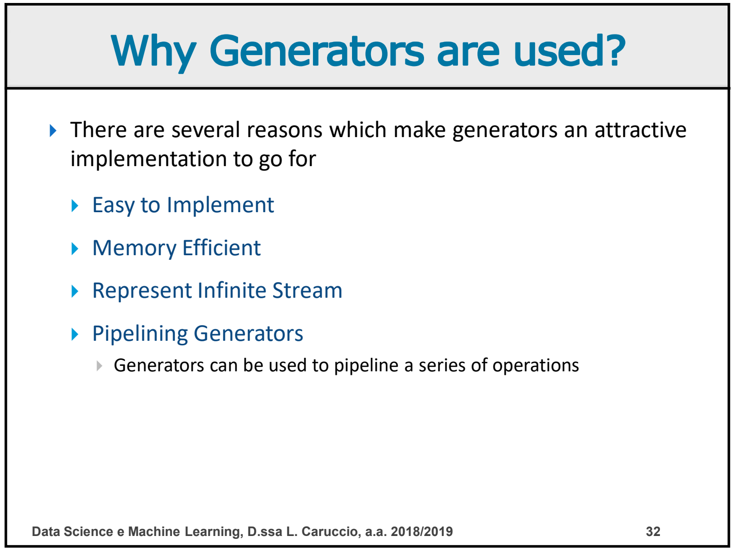


Fibonacci Generator

```
x=10 # Global Variable

# Simple Fibonacci generator
def fib(n):
    a, b = 0, 1
    for i in range(n):
        yield a
        a, b = b, a+b

# main
print(list(fib(x)))
```

Fibonacci	Arithmetic
0	
1	
1	0 + 1 = 1
2	1 + 1 = 2
3	1 + 2 = 3
5	2 + 3 = 5
8	3 + 5 = 8
13	5 + 8 = 13
21	8 + 13 = 21
34	13 + 21 = 34

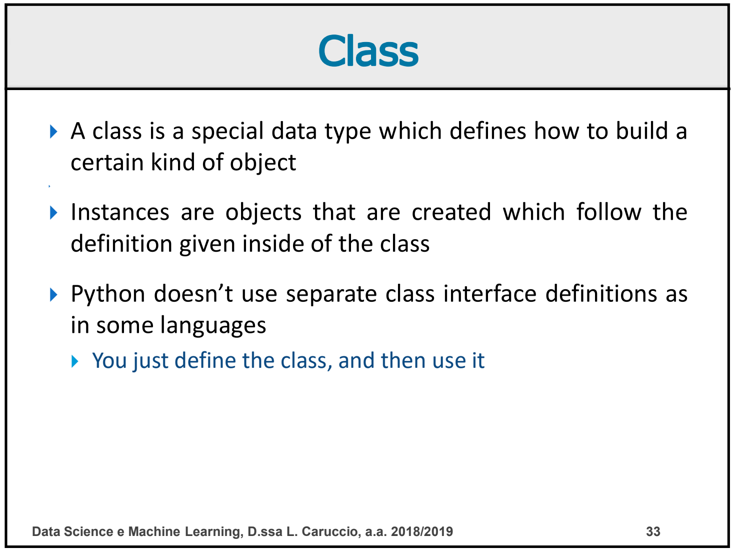
31



Why Generators are used?

- There are several reasons which make generators an attractive implementation to go for
  - Easy to Implement
  - Memory Efficient
  - Represent Infinite Stream
  - Pipelining Generators
    - Generators can be used to pipeline a series of operations

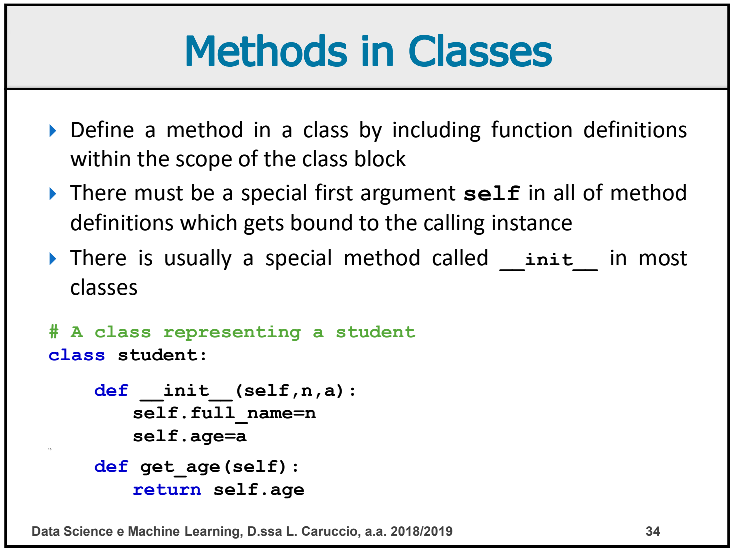
32



Class

- A class is a special data type which defines how to build a certain kind of object
- Instances are objects that are created which follow the definition given inside of the class
- Python doesn’t use separate class interface definitions as in some languages
  - You just define the class, and then use it

33

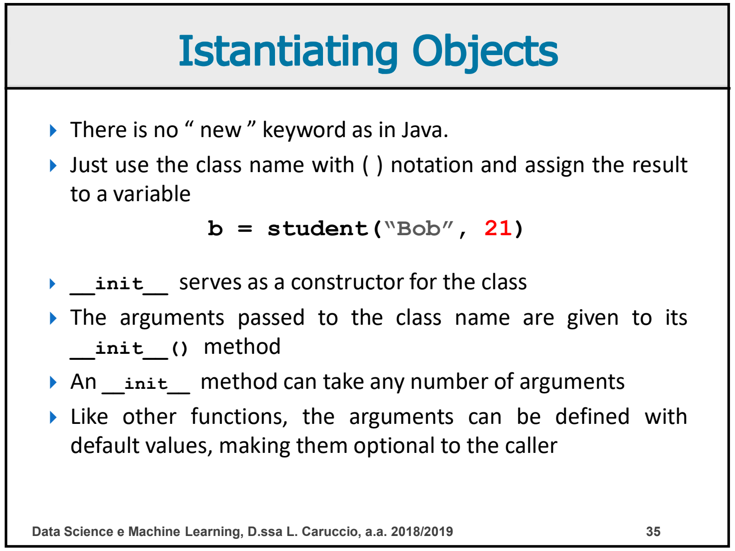


Methods in Classes

- Define a method in a class by including function definitions within the scope of the class block
- There must be a special first argument **self** in all of method definitions which gets bound to the calling instance
- There is usually a special method called __init__ in most classes

```
# A class representing a student
class student:

    def __init__(self,n,a):
        self.full_name=n
        self.age=a

    def get_age(self):
        return self.age
```

34

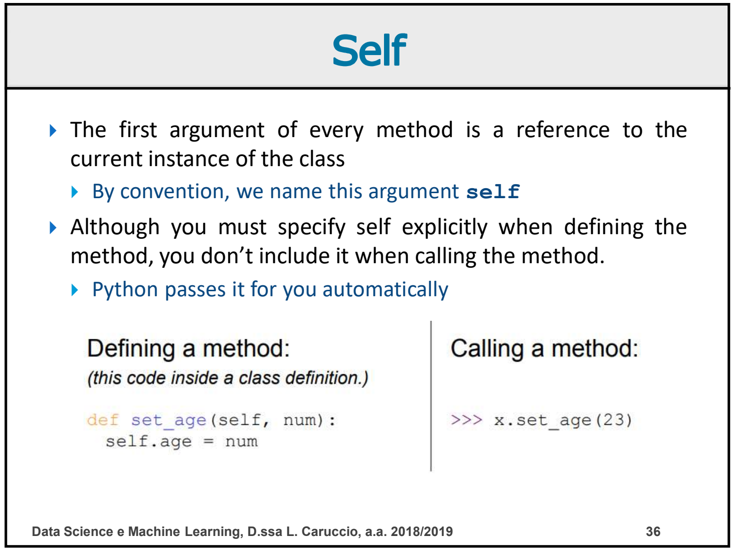


Istantiating Objects

- There is no “ new ” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable

```
b = student(“Bob”, 21)
```

- `__init__` serves as a constructor for the class
- The arguments passed to the class name are given to its `__init__()` method
- An `__init__` method can take any number of arguments
- Like other functions, the arguments can be defined with default values, making them optional to the caller



Self

- The first argument of every method is a reference to the current instance of the class
  - By convention, we name this argument **`self`**
- Although you must specify self explicitly when defining the method, you don’t include it when calling the method.
  - Python passes it for you automatically

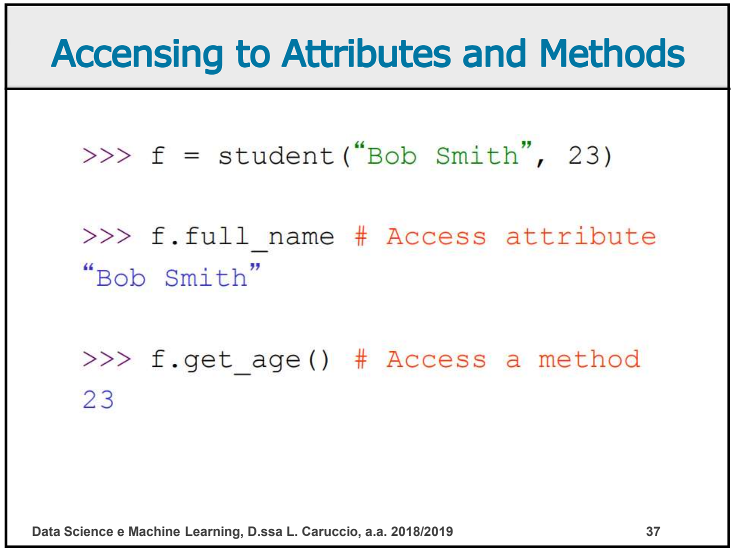
Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

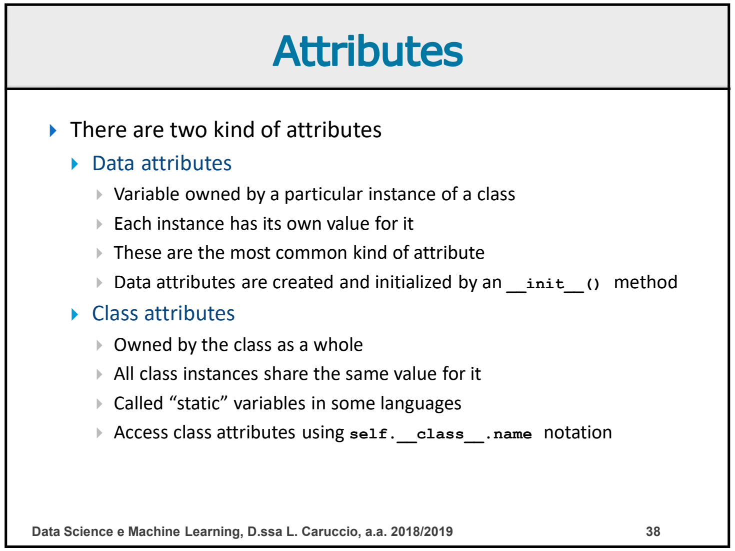
Calling a method:

```
>>> x.set_age(23)
```



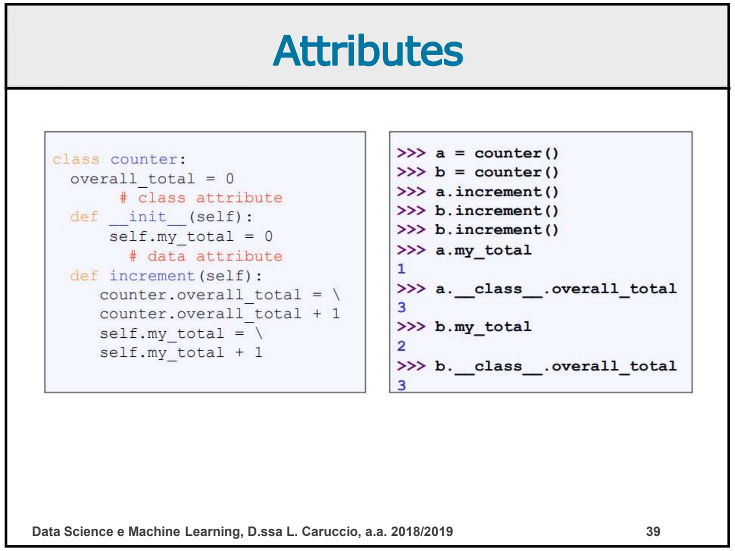
Accensing to Attributes and Methods

```
>>> f = student(“Bob Smith”, 23)

>>> f.full_name # Access attribute
“Bob Smith”

>>> f.get_age() # Access a method
23
```



Attributes

- There are two kind of attributes
  - Data attributes
    - Variable owned by a particular instance of a class
    - Each instance has its own value for it
    - These are the most common kind of attribute
    - Data attributes are created and initialized by an `__init__()` method
  - Class attributes
    - Owned by the class as a whole
    - All class instances share the same value for it
    - Called “static” variables in some languages
    - Access class attributes using `self.__class__.name` notation

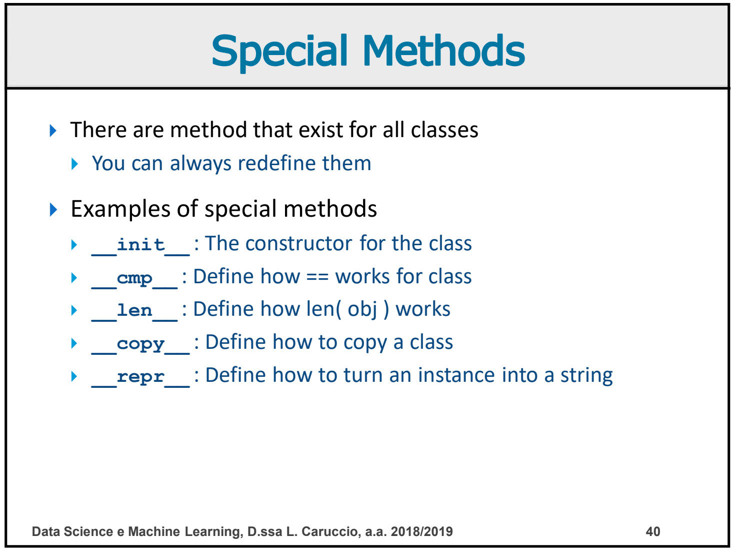


Attributes

```
class counter:
  overall_total = 0
       # class attribute
  def __init__(self):
      self.my_total = 0
        # data attribute
  def increment(self):
     counter.overall_total = \
     counter.overall_total + 1
     self.my_total = \
     self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

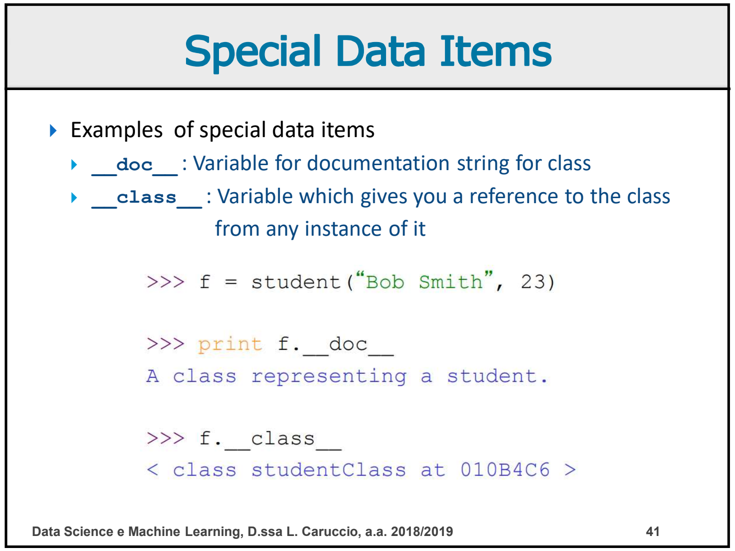
39



Special Methods

- There are method that exist for all classes
  - You can always redefine them
- Examples of special methods
  - **`__init__`**: The constructor for the class
  - **`__cmp__`**: Define how == works for class
  - **`__len__`**: Define how len(obj) works
  - **`__copy__`**: Define how to copy a class
  - **`__repr__`**: Define how to turn an instance into a string

40

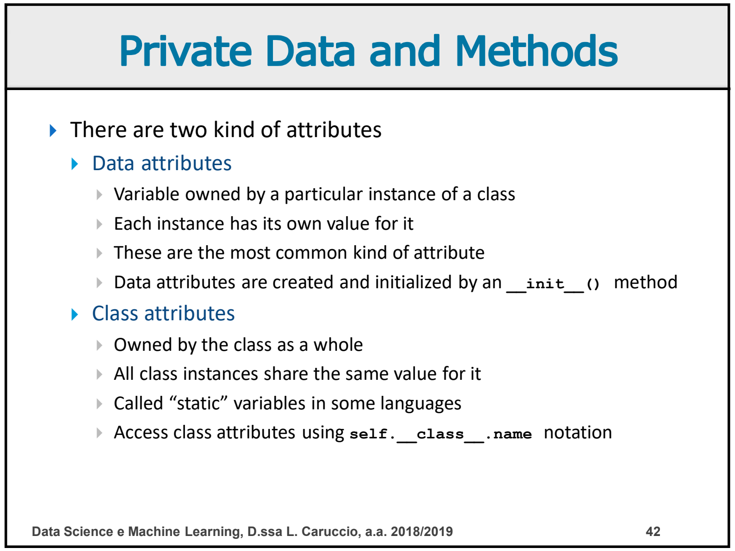


Special Data Items

- Examples of special data items
  - **`__doc__`**: Variable for documentation string for class
  - **`__class__`**: Variable which gives you a reference to the class from any instance of it

```
>>> f = student(“Bob Smith”, 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >
```

41



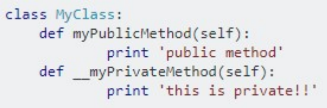
Private Data and Methods

- There are two kind of attributes
  - Data attributes
    - Variable owned by a particular instance of a class
    - Each instance has its own value for it
    - These are the most common kind of attribute
    - Data attributes are created and initialized by an **`__init__()`** method
  - Class attributes
    - Owned by the class as a whole
    - All class instances share the same value for it
    - Called “static” variables in some languages
    - Access class attributes using **`self.__class__.name`** notation

42

Private Data and Methods

- Any attribute/method with two leading underscores in its name (but none at the end) is private and can't be accessed outside of class
- Note: Names with two underscores at the beginning and the end are for built-in methods or attributes for the class Variable owned by a particular instance of a class



```
class MyClass:
    def myPublicMethod(self):
            print 'public method'
    def __myPrivateMethod(self):
            print 'this is private!!'
```

Final Tips (1)

- Import a module

```
# Import all functions from a module
from module_name import *

# Import a specific function from a module
from module_name import function_name
```

Final Tips (2)

- Exception Handling

```
# Below code will open a file
fName="vechicles.txt"
try:
   with open(fName, 'r') as f
   print(f.readline())
except IOError as e:
   print('IO Error')
finally:
   print('File has been closed')
```