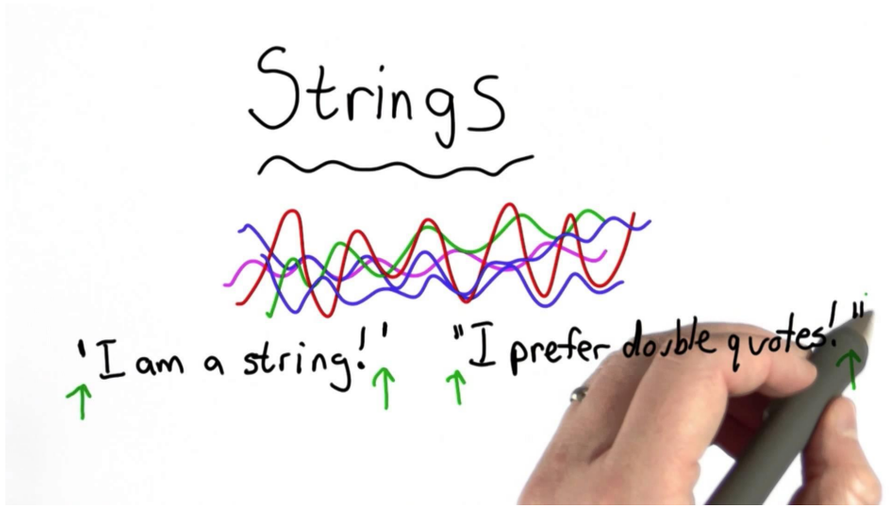
Programming For Data Science

Part Nine:

Strings again!

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it



- You know how to perform basic operations on strings
- You know format and len()
- You know how to use strings as collections of characters
    - for letter in “Python”:

\

- The \ character can be used as escape
- Also, at the end of a line, it tells the Python's interpreter that the line continues on the next line

```
a = 2 + 7 - \
    7 + 2 \
-4


long = "I'm here in class \
waiting for more instructions!"
```

Multiline strings

- Use triple double or single quotes

```
long = “‘I’m here in class
waiting for more instructions!’”
```

- The string `long` is cut at the end of the line
  - A newline character `\n` is inserted automatically
  - Not interpreted as a comment, since assigned to a variable

\n

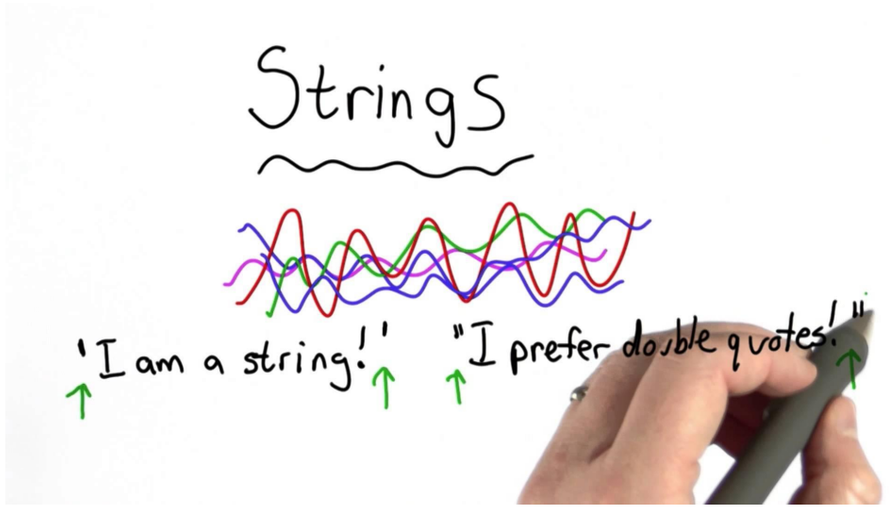
- It can be used to explicitly tell Python to cut a line

```
long = “I’m here in class \n\
waiting for more instructions!”
```

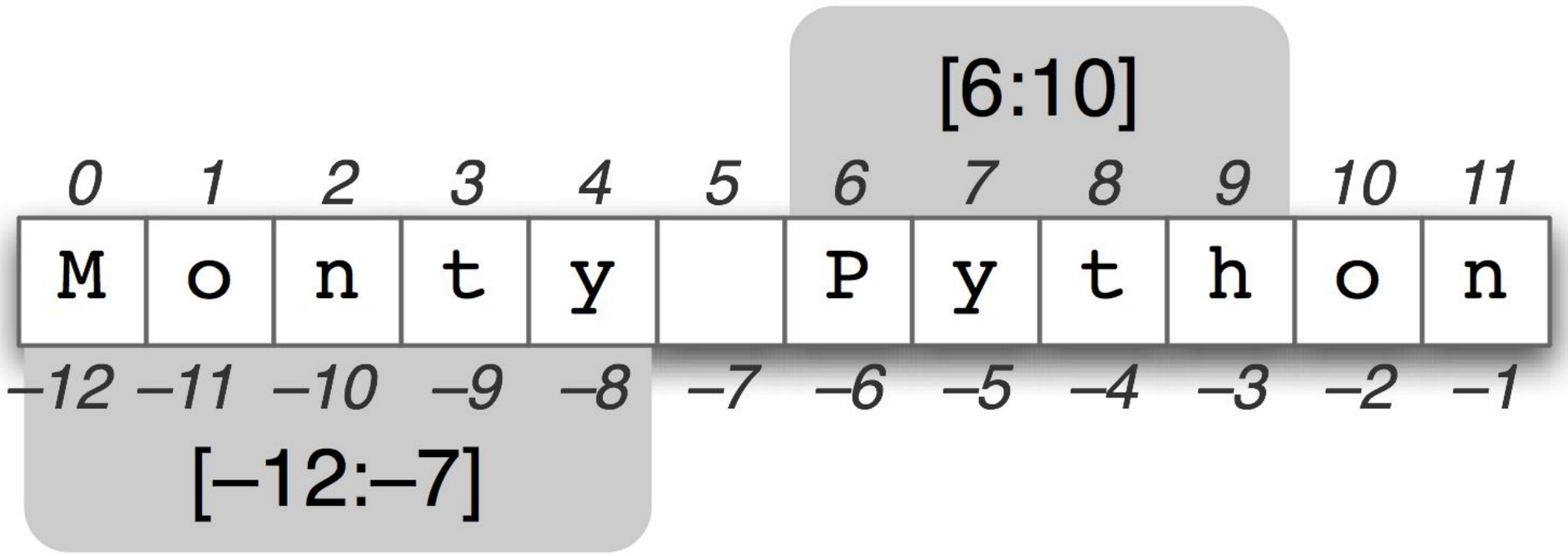
- Even if there is \ at the end of the line, \n cuts it anyway

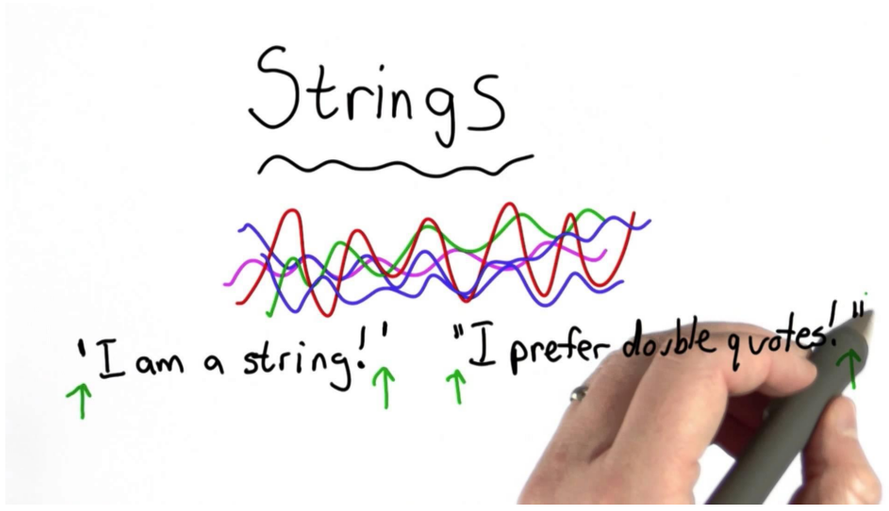
Escape sequences

- \ followed by a code is called an *escape sequence*
- \n, \', \", \\
- \t —> tabulation
- \xnn, with nn an hexadecimal number
    - \x20—> (32 in decimal) space character

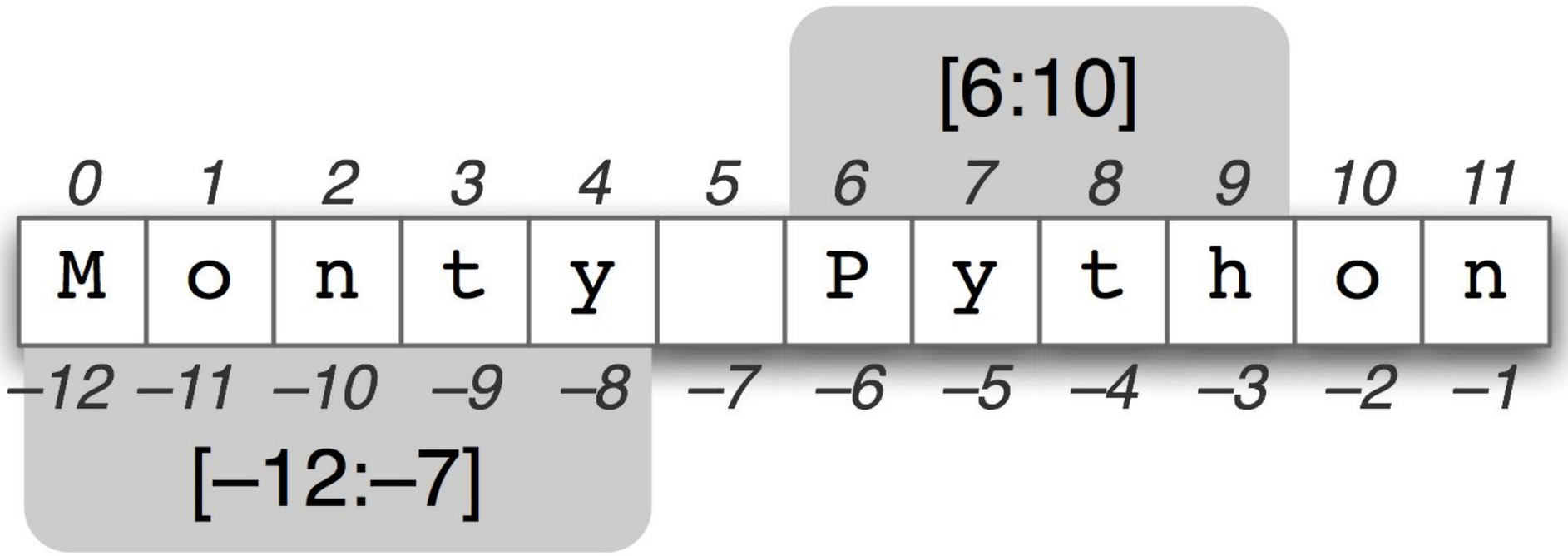


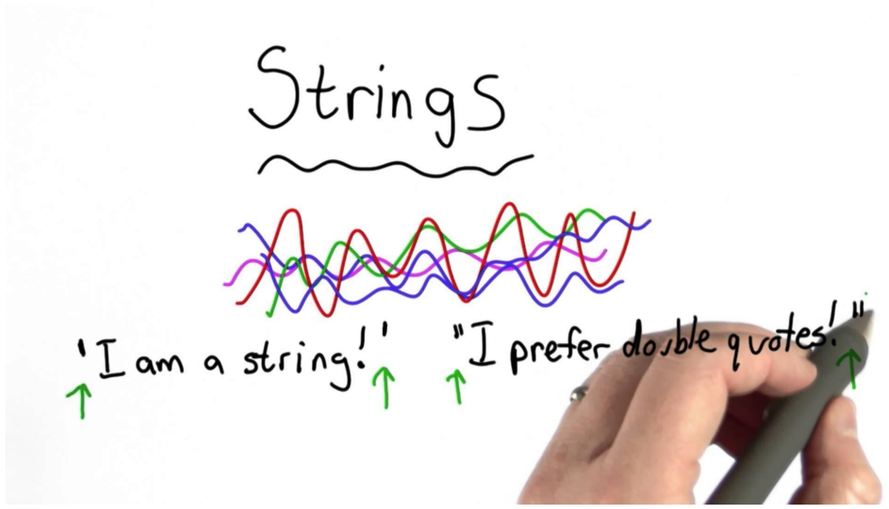
- Strings are collections of characters, and each letter can be accessed by its position in the string





- The last letter of a string has index len(s)-1
- The first letter of a string has index either 0 or -len(s)

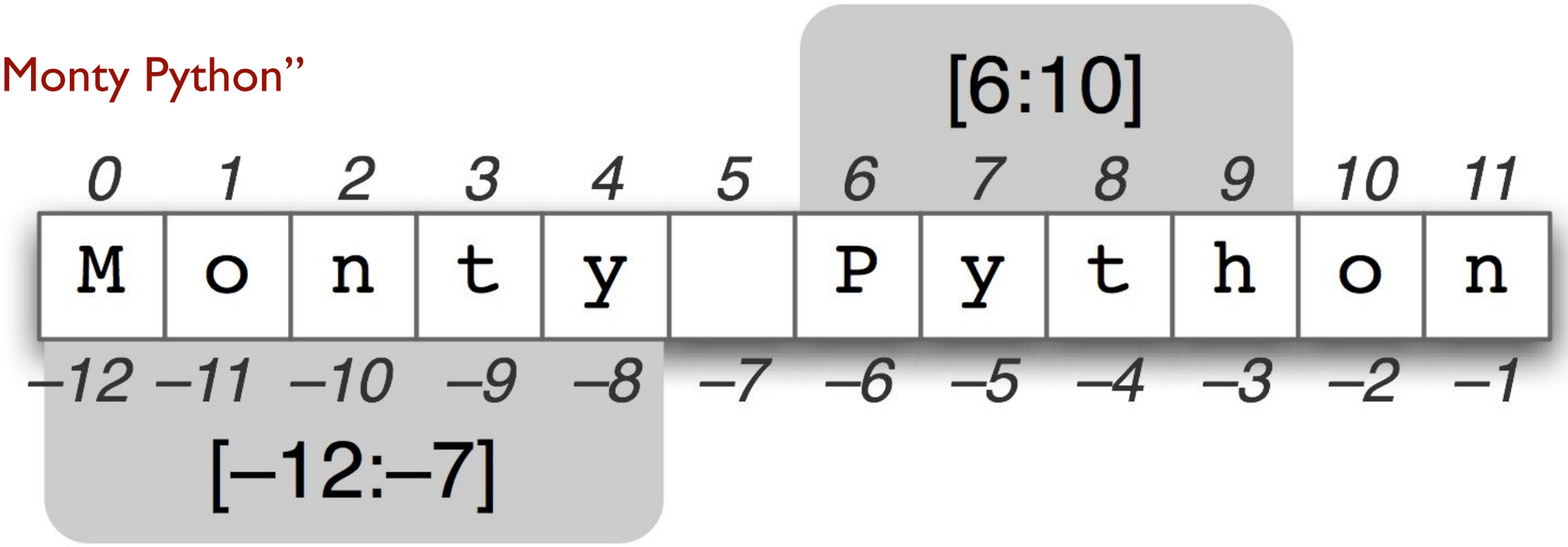


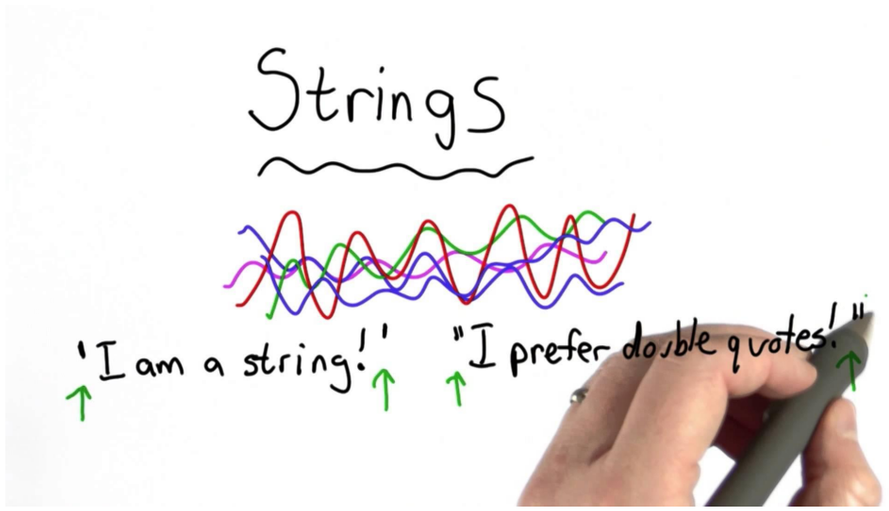


index can be the result of any calculation

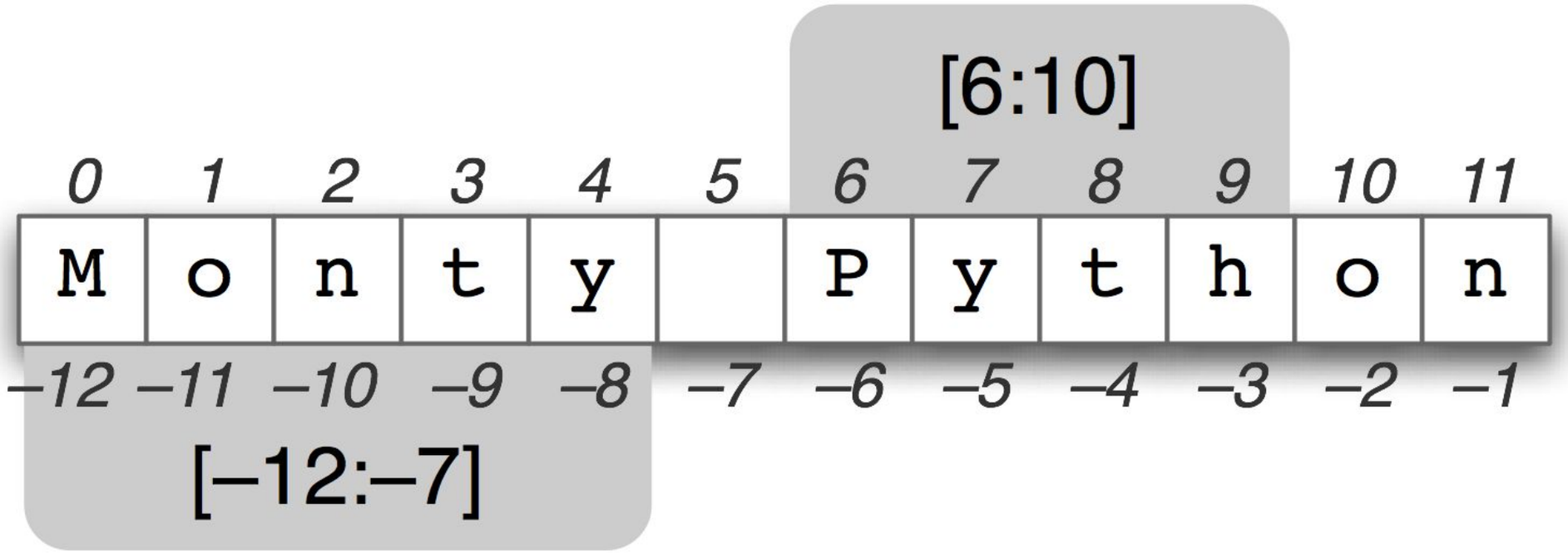
- Character of a string can be accessed using string[index] syntax
  - M == string[0] or M == string[-len(string)]

string = “Monty Python”

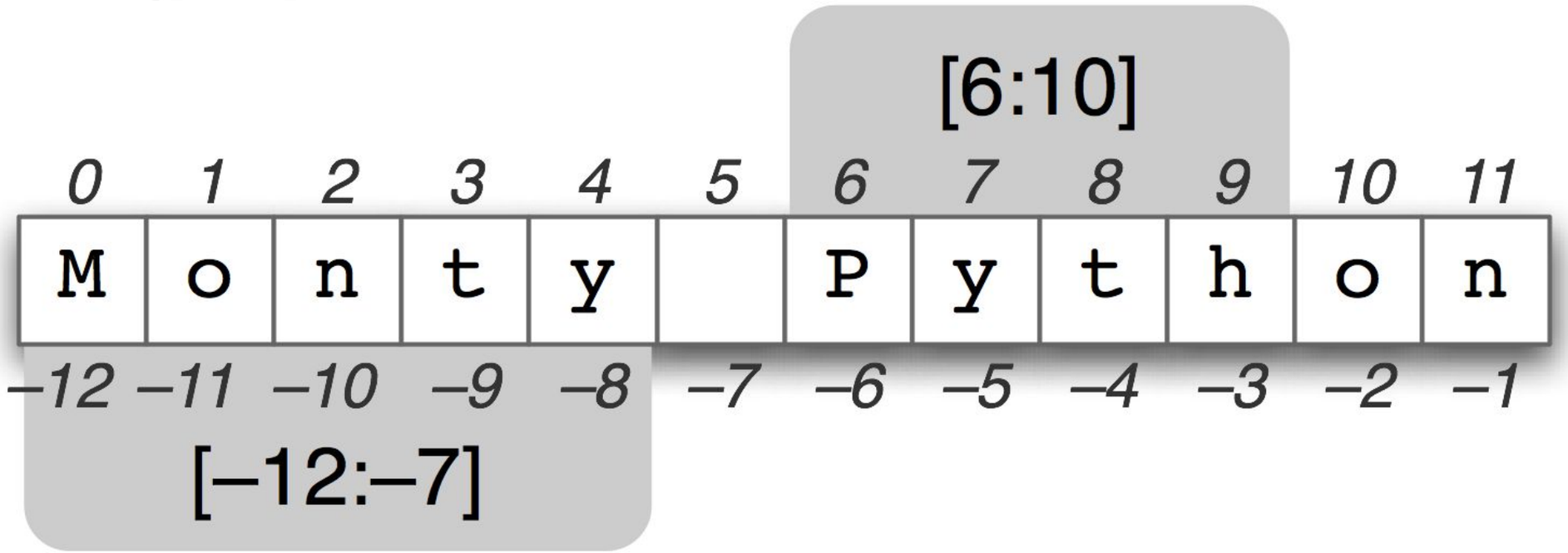




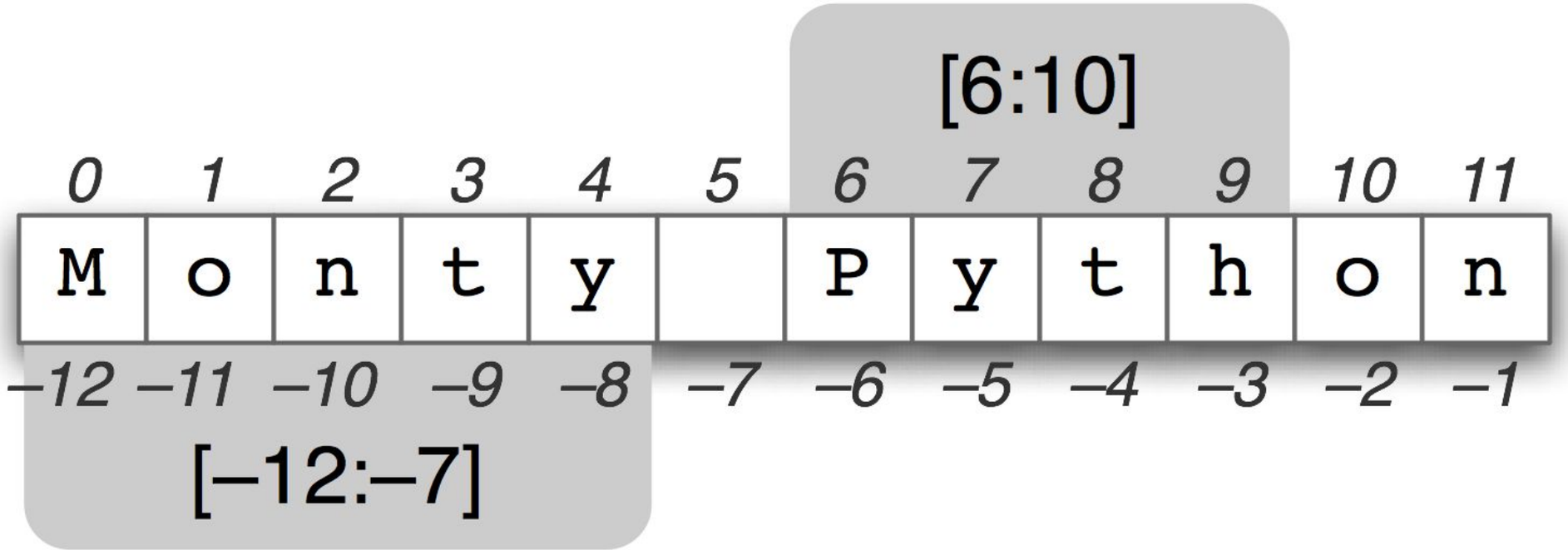
- A slice is a sub-string, denoted by using :
  - “Pyth” == string[6:10]
  - “Monty” == string[-12:-8]



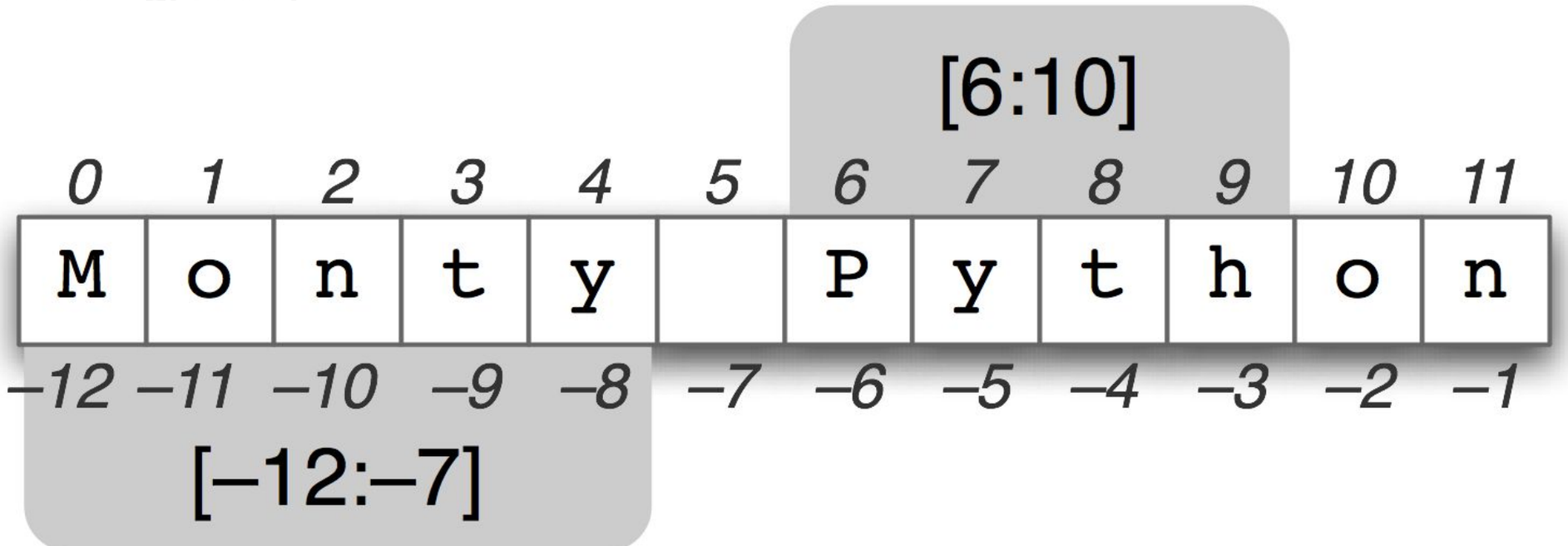
- string[:] —> ????
- string[:5] —> ????
- string[0:] —> ????
- string[:1000] —>????
- string[1:-1] —> ????
- string[1000] —> ????



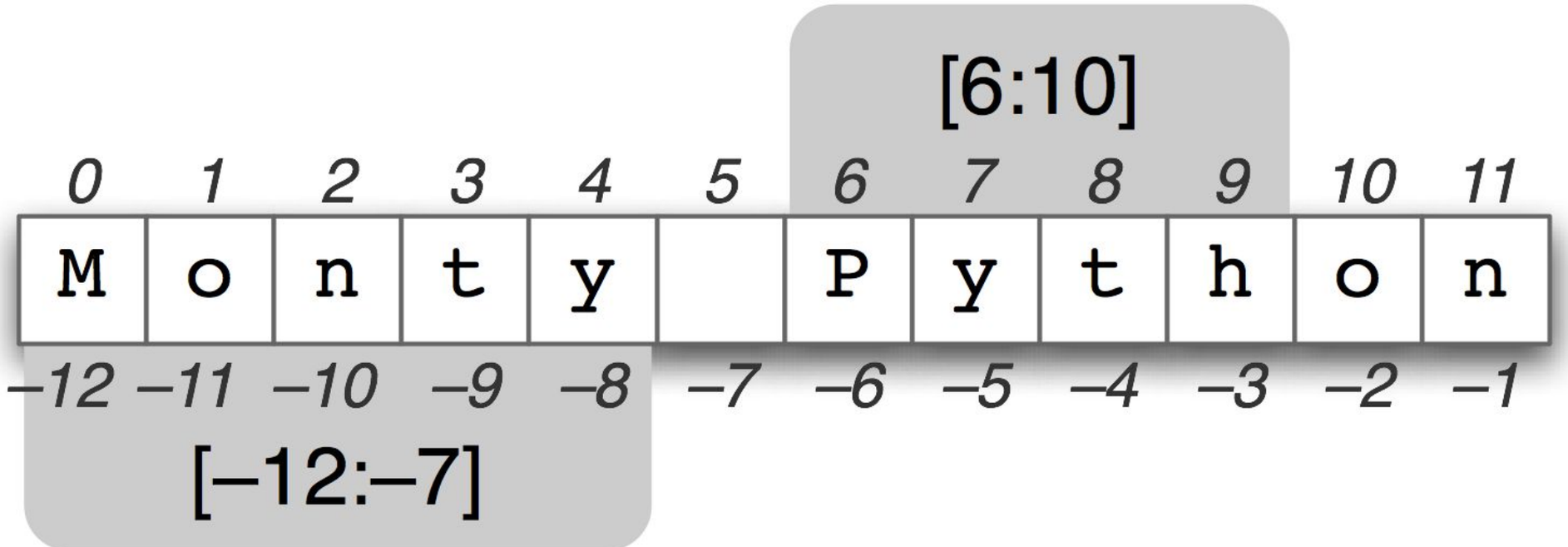
- string[:] —> Monty Python
- string[:5] —> ????
- string[0:] —> ????
- string[:1000] —>????
- string[1:-1] —> ????
- string[1000] —> ????



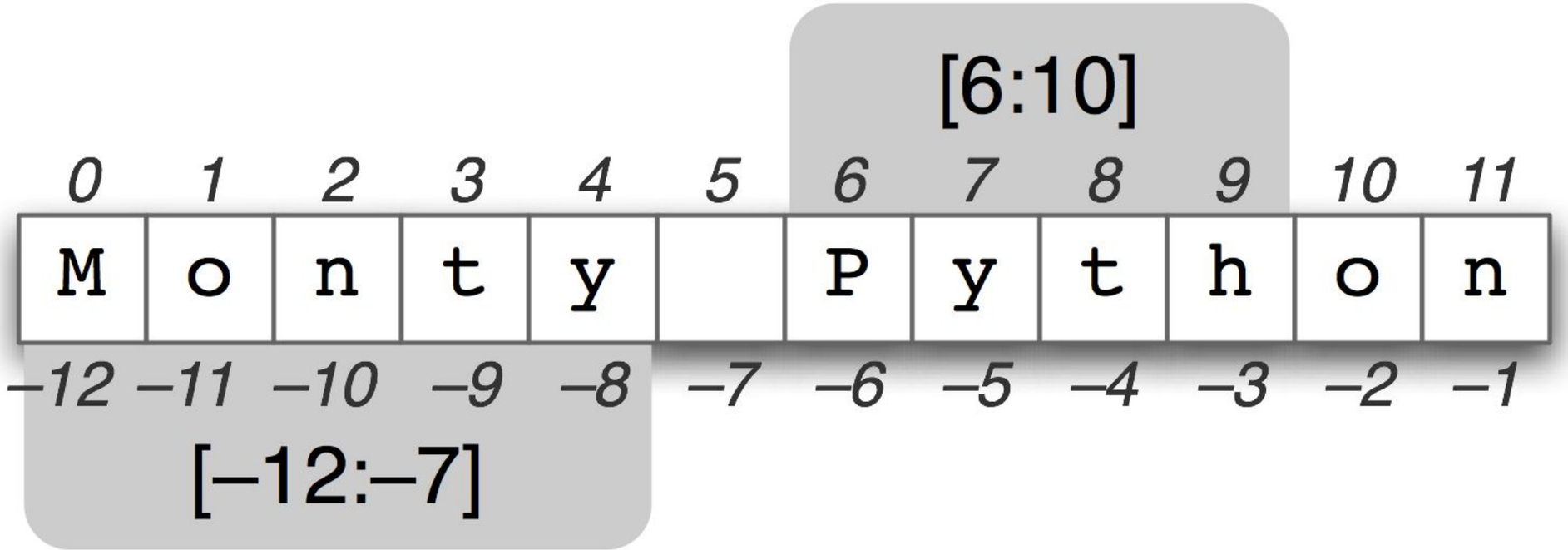
- string[:] —> Monty Python
- string[:5] —> Monty
- string[0:] —> ????
- string[:1000] —>????
- string[1:-1] —> ????
- string[1000] —> ????



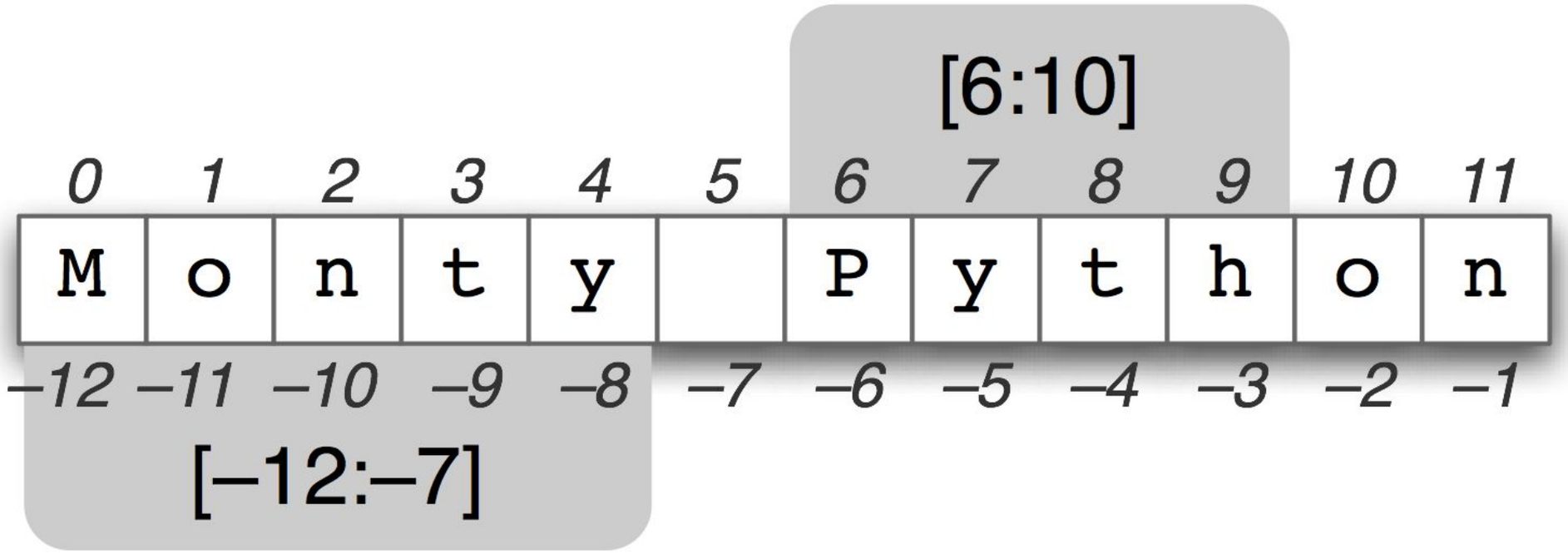
- string[:] —> Monty Python
- string[:5] —> Monty
- string[0:] —> Monty Python
- string[:1000] —>????
- string[1:-1] —> ????
- string[1000] —> ????



- string[:] —> Monty Python
- string[:5] —> Monty
- string[0:] —> Monty Python
- string[:1000] —> Monty Python
- string[1:-1] —> ????
- string[1000] —> ????

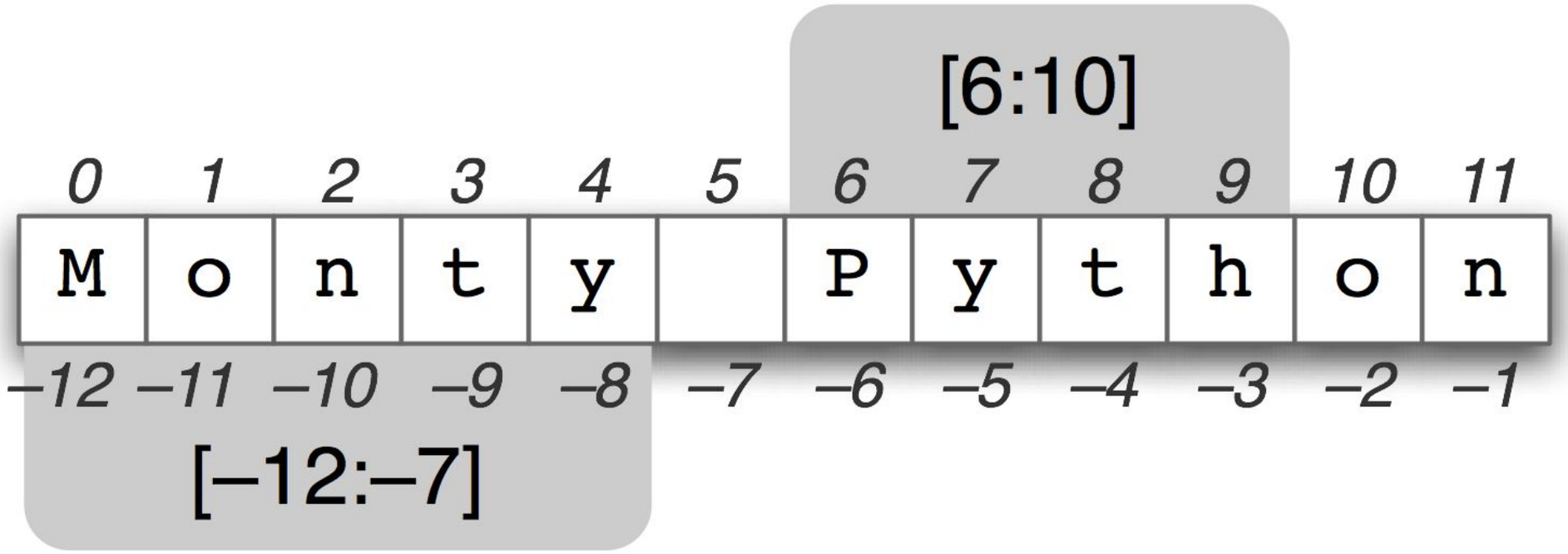


- string[:] —> Monty Python
- string[:5] —> Monty
- string[0:] —> Monty Python
- string[:1000] —> Monty Python
- string[1:-1] —> onty Pytho
- string[1000] —> ????



- string[:] —> Monty Python
- string[:5] —> Monty
- string[0:] —> Monty Python
- string[:1000] —> Monty Python
- string[1:-1] —> onty Pytho
- string[1000] —> IndexError

No IndexError

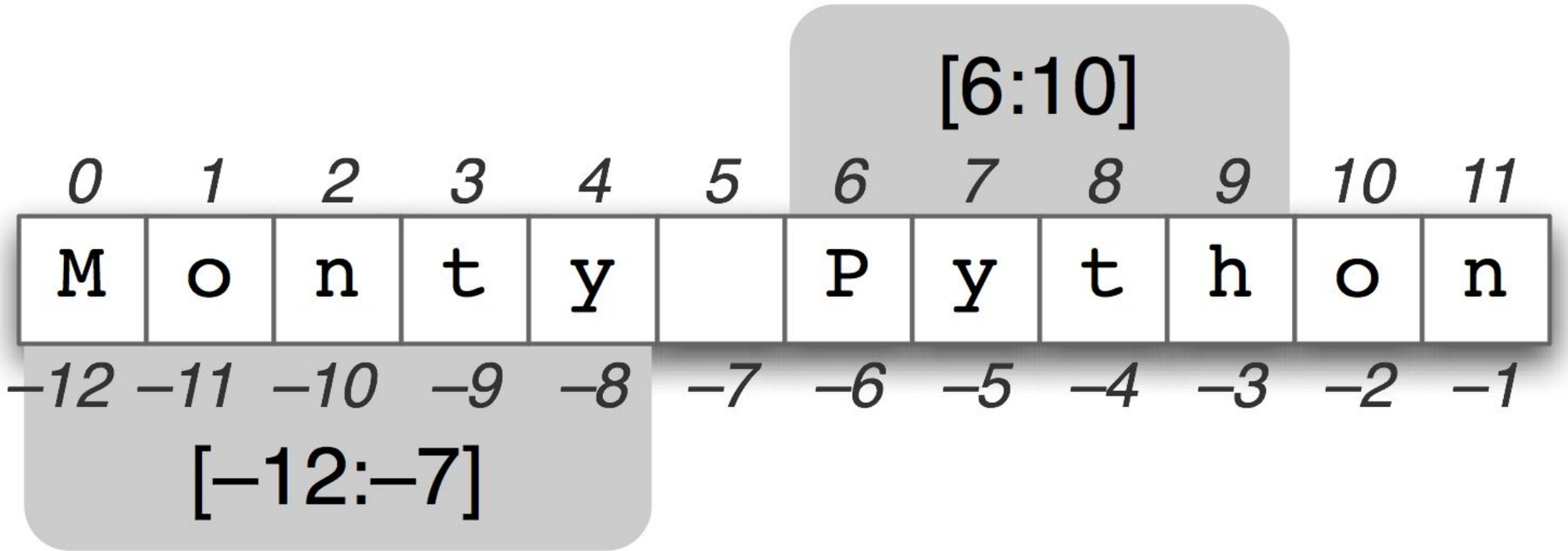


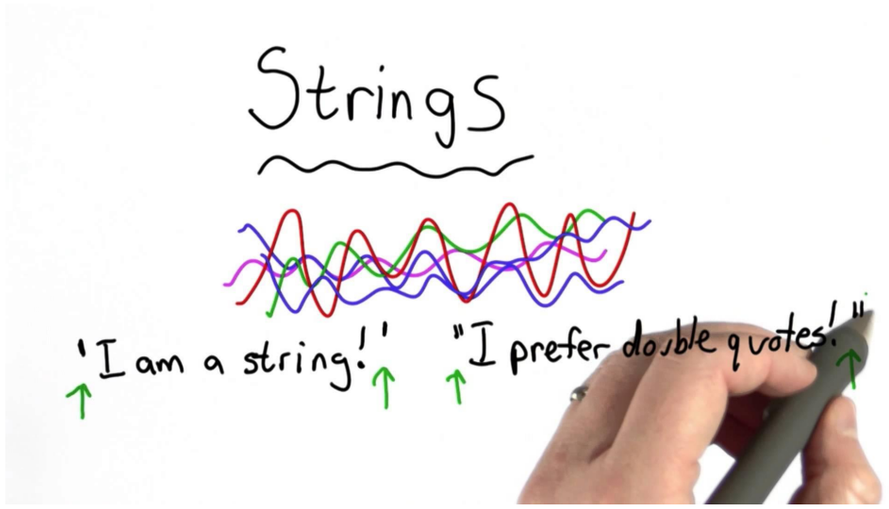
Extended slice

- A slice can take a third parameter, denoting the step
  - string[begin:end:step]

```
string = “Monty Python”
print(string[::2])
```

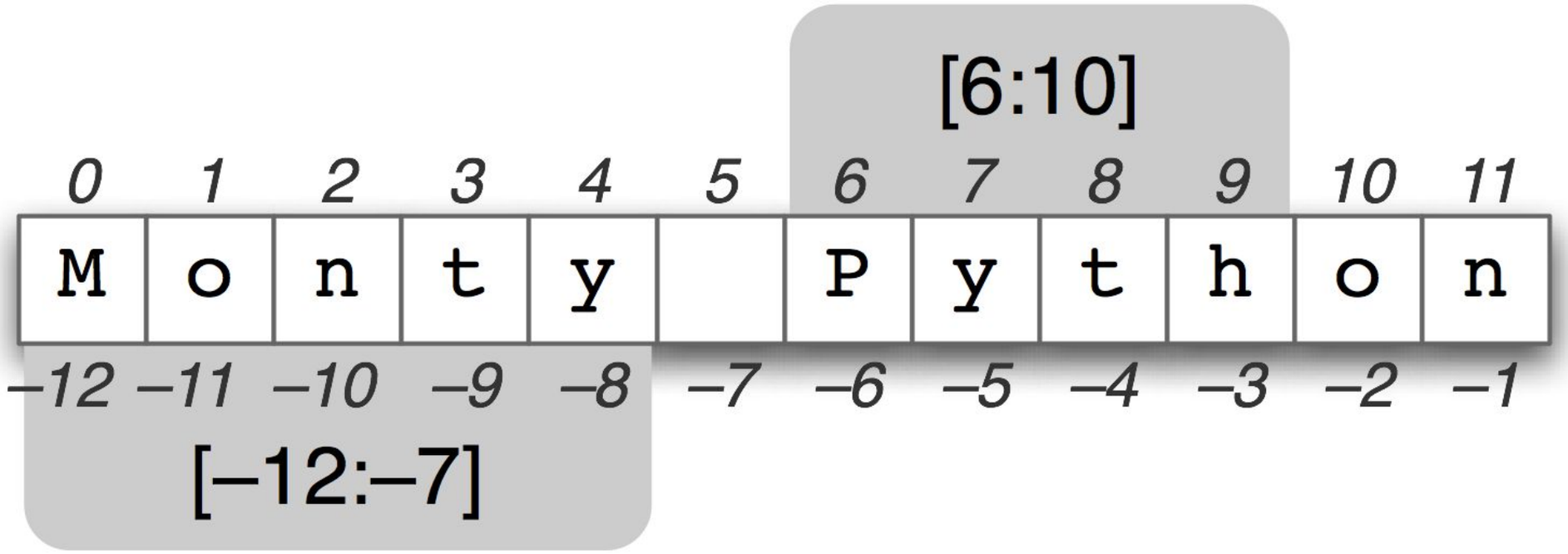
MnyPto

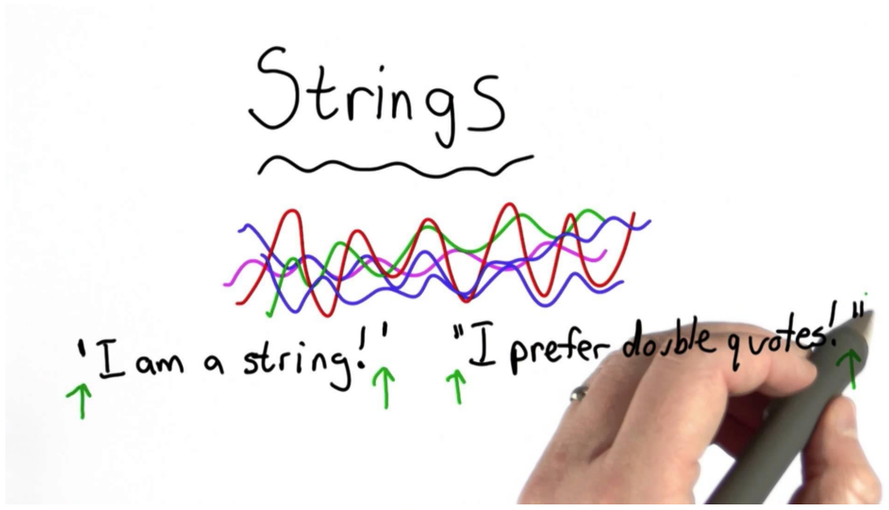





```
string = “Monty Python”
for i in range (11, -1, -1):
    print(string[i])
```

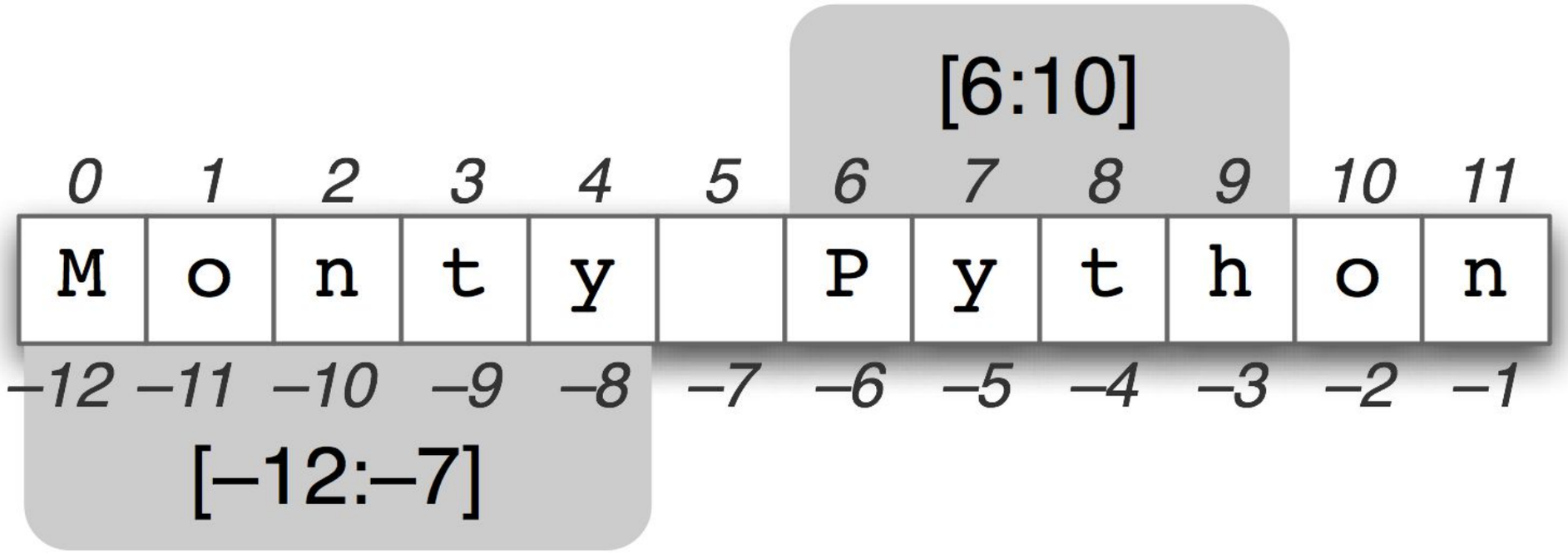
????

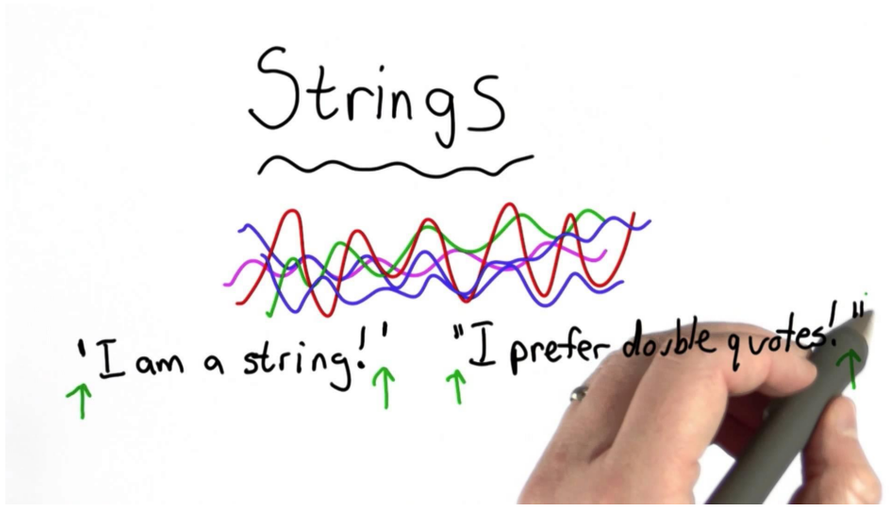





```
string = “Monty Python”
for i in range (11, -1, -1):
    print(string[i])
```

Reverse a string

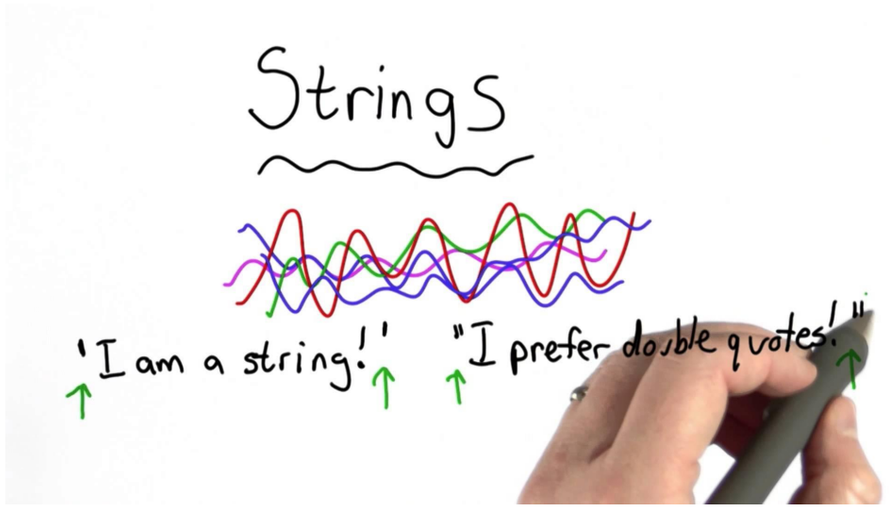




- Strings are immutable

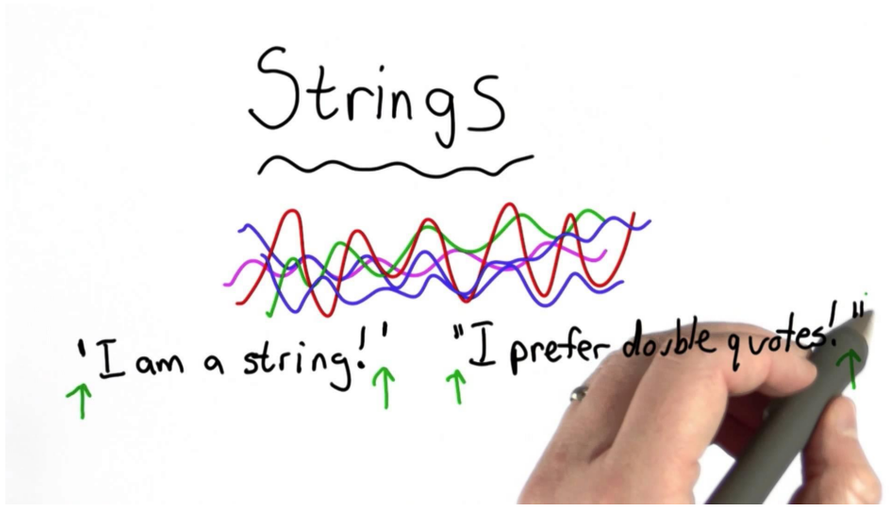
```
string = “Mocty Python”

string[2] = “n”
```

Runtime error!



- How can I modify string?

```
string = “Mocty Python”
????
```



- How can I modify string?

```
string = “Mocty Python”
string = string[:2] + “n” + string[3:]
```

Useful methods for strings

- s.strip()
  - Removes from a string leading and trailing whitespaces
- s.upper() and s.lower()
  - Creates a version of a string of which all letters are capitals / lower case
- s.find(substring, index) — index is optional
  - Can be used to search in a string from the starting index of a particular substring
  - Returns the lowest index where substring starts, or -1
- s.replace(lookFor, substituteWith)
  - Replaces all occurrences of a lookFor in s with substituteWith
- s.split(sep) — if no sep, white space is used
  - Splits a string up in words, based on a given character or substring which is used as separator — returns a list of words
- s.join(list) — s is the separator, and list the words to join
  - Joins a list of words together, separated by a specific separator

Encoding — ASCII

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	@	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(	72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29	)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[	123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	|
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D	]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

7-bits code (128 characters)

ord(“A”) = 65

chr(97) = “a”

Used to compare strings

orange < ordinary

Encoding — UTF-8

- Python supports Unicode, that supports for more characters, and in particular UTF-8
    - Can represent ‘strange’ characters and letters
- To display a Unicode character, use its code
    - \uxxxx, where xxxx is a hex number
        - \u0391 is the α Greek character
- Wherever possible, use ASCII
    - Don’t use UTF-8 characters for variables’ and functions’ names!!!!

(Suggested) Exercises (1)

- Write code that for a string prints the indices of all of its vowels (a, e, i, o, and u). This can be done with a `for` loop or a `while` loop, though the while loop is more suitable
- Write code that uses two strings. For each character in the first string that has exactly the same character at the same index in the second string, you print the character and the index. Watch out that you do not get an “index out of bounds” runtime error. Test it with the strings "The Holy Grail" and "Life of Brian"
- Write a function that takes a string as argument, and creates a new string that is a copy of the argument, except that every non-letter is replaced by a space (e.g., "ph@t l00t" is changed to "ph t l t"). To write such a function, you will start with an empty string, and traverse the characters of the argument one by one. When you encounter a character that is acceptable, you add it to the new string. When it is not acceptable, you add a space to the new string. Note that you can check whether a character is acceptable by simple comparisons. For example, any lower case letter can be found using the test if >= 'a' and <= ‘z':

(Suggested) Exercises (2)

- **Pig Latin** is a language game, where you move the first letter of the word to the end and add "ay." So "Python" becomes "ythonpay." To write a Pig Latin translator in Python, here are the steps we'll need to take:
    - Ask the user to input a word in English
    - Make sure the user entered a valid word
    - Convert the word from English to Pig Latin
    - Display the translation result

Write and test a function that implements the a Pig Latin translator

(Suggested) Exercises (3)

- In the string “How much woot would a wootchuck chuck if a wootchuck could chuck the woot” the word "wood" is misspelled. Use replace() to replace all occurrences of this spelling error with the correct spelling
- Write a program that prints a “cleaned” version of all the words in a string. Everything that is not a letter should be removed and be considered a separator. All the letters should be lower case. For example, the string "I'm sorry, sir." should produce four words, namely "i", "m", "sorry", and "sir"
- **All** exercises in Chapter 10 of the reference book

Programming For Data Science

Part Ten:

Tuples

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Tuples

- One or more values, separated by commas
  - With or without () parentheses

```
t1 = “pippo”, “pluto”
t2 = (“qui”, “quo”, “qua”)
t3 = (t1, t2)
print(len(t2))
print(len(t3))
```

Tuples

- One or more values, separated by commas
  - With or without () parentheses
  - You can mix data types

```
t1 = 1, “pluto”
t2 = (3.14, 42, “numbers”)
t3 = (t1, t2)
print(len(t2)
print(len(t3))
```

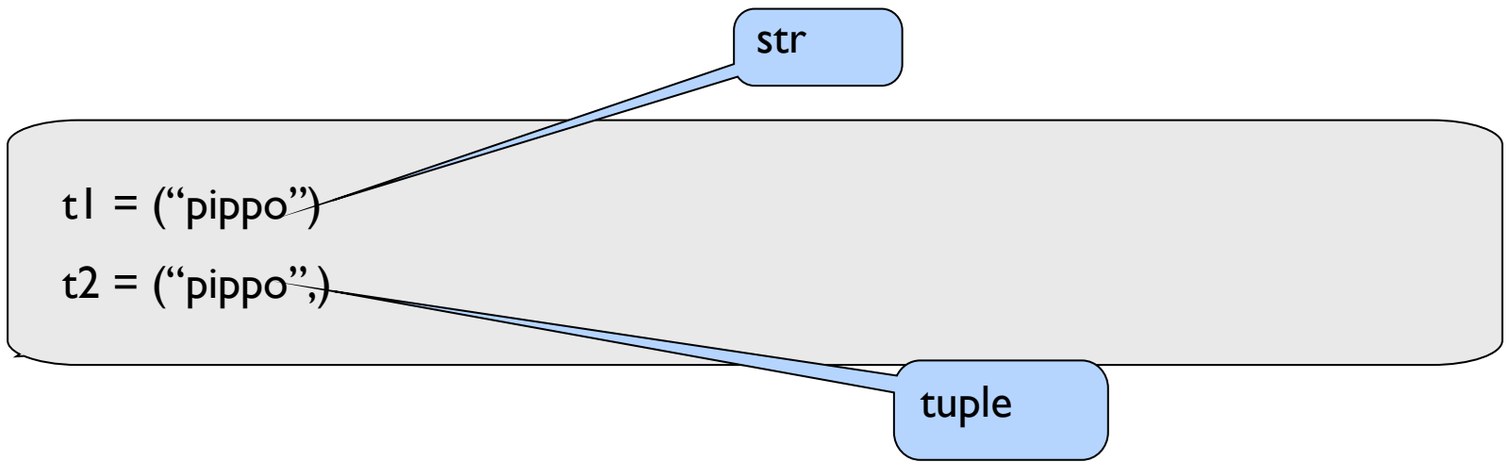
Tuples

- One or more values, separated by commas
  - With or without () parentheses
  - You can mix data types
  - You can access elements using for

```
t2 = (3.14, 42, 88)
for element in t2:
    print (element)
print (max(t2), min(t2), sum(t2))
```

Tuples

- Since parenthesis are optional, one value is not a tuple
  - To have a tuple, add a comma



Multiple assignments

```
a, b = “pippo”, 5

t1, t2 = (“pippo”, “pluto”), 4
```

Multiple assignments

```
a, b = “pippo”, 5
t1, t2 = (“pippo”, “pluto”), 4
a, b = 5, 6, 7
```

ValueError

Tuples

- You can access tuple's values by its index

```
t2 = (3.14, 42, 88)
print (t2[0])
print (t2[1:3])
i=0;
while i < len(t2):
    print( t2[i] )
    i += 1
```

Comparing tuples

- You can compare tuples, using the usual comparing operators

t1 = (3.14, 1, 42) Equal: go on! t2 = (3.14, 1, 2)

t1 = (3.14, 1, 42) Equal: go on! t2 = (3.14, 1, 2)

t1 = (3.14, 1, 42) t1 is bigger! t2 = (3.14, 1, 2)

Tuples cannot be changed

- ….like strings!

```
t1 = (“pippo”, “pluto”)
t1[0]= “paperone”
```

Error

Exercises

- **All** exercises in Chapter 11 of the reference book

Programming For Data Science

Part Eleven:

Lists

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Lists

A **list** is a collection of elements. The elements are *ordered*.

Lists

A **list** is a collection of elements. The elements are *ordered*.

```
mesi = [ “Gennaio”, “Febbraio”, “Marzo”, “Aprile”, “Maggio”, “Giugno”,\
        “Luglio”, “Agosto”, “Settembre”, “Ottobre”, “Novembre”, “Dicembre” ]


m = int(input(“m:”))
while 1 <= m <= 12:
   print (mesi[m -1])
   m = int(input(“m:”))
```

Lists

A **list** is a collection of elements. The elements are *ordered*.

```
mesi = [ “Gennaio”, “Febbraio”, “Marzo”, “Aprile”, “Maggio”, “Giugno”,\
        “Luglio”, “Agosto”, “Settembre”, “Ottobre”, “Novembre”, “Dicembre” ]


m = int(input(“m:”))
while 1 <= m <= 12:
    print (mesi[m -1])
    m = int(input(“m:”))
```

mesi[i] accesses the i-th element.

Starting from 0!

Lists

A **list** is a collection of elements. The elements are *ordered*.

```
mesi = [ “Gennaio”, “Febbraio”, “Marzo”, “Aprile”, “Maggio”, “Giugno”,\
        “Luglio”, “Agosto”, “Settembre”, “Ottobre”, “Novembre”, “Dicembre” ]


i = 0
while i < len(mesi):
   print (mesi[i])
   i += 1
```

Displays the elements of the list, one per line

Lists

A **list** is a collection of elements. The elements are *ordered*.

```
mesi = [ “Gennaio”, “Febbraio”, “Marzo”, “Aprile”, “Maggio”, “Giugno”,\
        “Luglio”, “Agosto”, “Settembre”, “Ottobre”, “Novembre”, “Dicembre” ]


i = len(mesi)-1
while 0 <= i:
   print (mesi[i])
   i -= 1
```

Displays the elements of the list in reverse order

Lists

A **list** is a collection of elements. The elements are *ordered*.

```
numeri = []

n = int(input("n: "))

i = 0
while i < n:
    numeri.append(input("successivo: "))
    i += 1

i = 0
while i < n:
    print (numeri[i])
```

Empty list

Insert in the list the value

Displays the elements, one per line

Displays the list [e1, e2, …]

Lists

- The operator + between two lists, merges them
  - You cannot add a new element to a list with the +

```
list1 = [“pippo”, “pluto”]
list2 = [“paperino”]
list3 = list1 + list2
```

Lists

A **list** is a collection of elements. The elements are *ordered*.

Example	What it does
`list[2]`	accesses element with index 2
`list[2] = 3`	sets to 3 second element
`del list[2]`	deletes second element
`len(list)`	list
`value in list`	s in the list
`value not in lis`	true if value is not in the list
`list.sort()`	
`list.index(value)`	returns index of first occurrence of value
`list.append(value)`	appends value to the end of the list

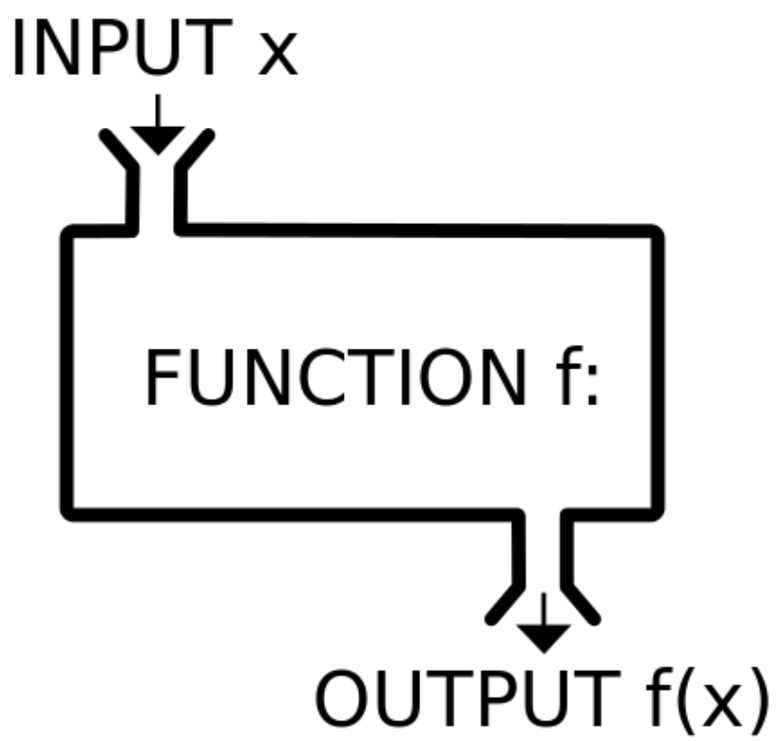
Function

Method

Lists are mutable!

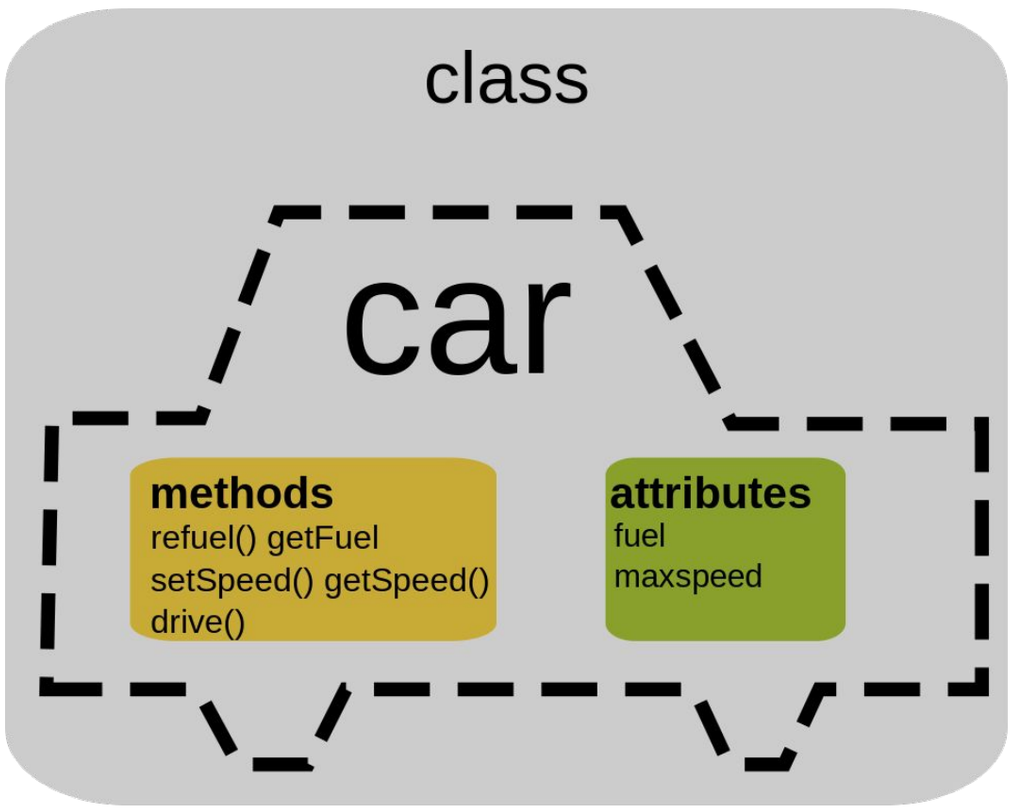
Function vs Method

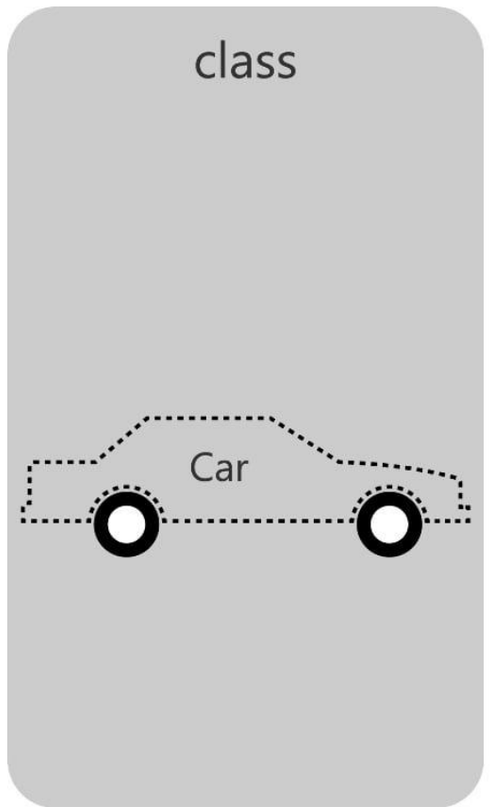
- A **function** is a piece of code that is called by name. It can be passed data to operate on (i.e. the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed

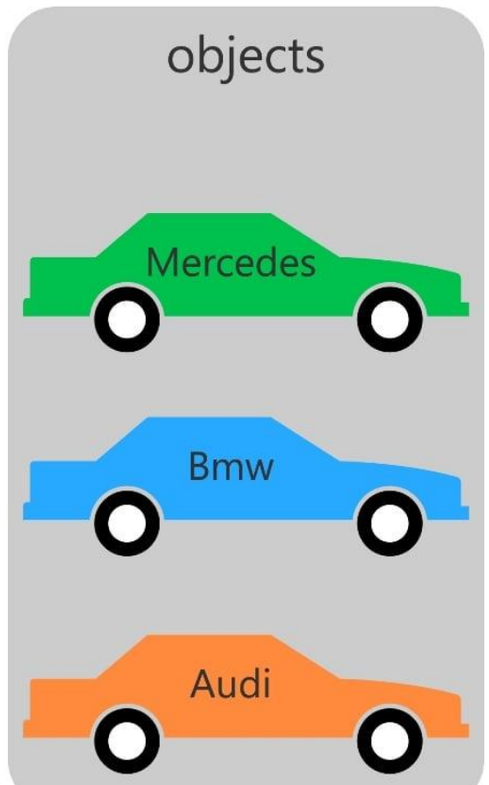


Function vs Method

- A **method** is a piece of code that is called by a name that is associated with a class/object







Function vs Method

- A **method** is a piece of code that is called by a name that is associated with a class
- In most respects it is identical to a function except for two key differences:
  - A method is implicitly passed the object on which it is called
  - A method is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data)

Lists

- You can assign list to variables that contain lists

```
list1 = [“pippo”, “pluto”]
list2 = [“quo”]
list2 = list1
list2[1] = “paperino”
print(list1)
```

What do we get here?

Aliasing

- You can assign list to variables that contain lists
  - You do not get a copy, but just an *alias pointing to the same list*

```
list1 = [“pippo”, “pluto”]
list2 = [“quo”]
list2 = list1
list2[1] = “paperino”
print(list1)
```

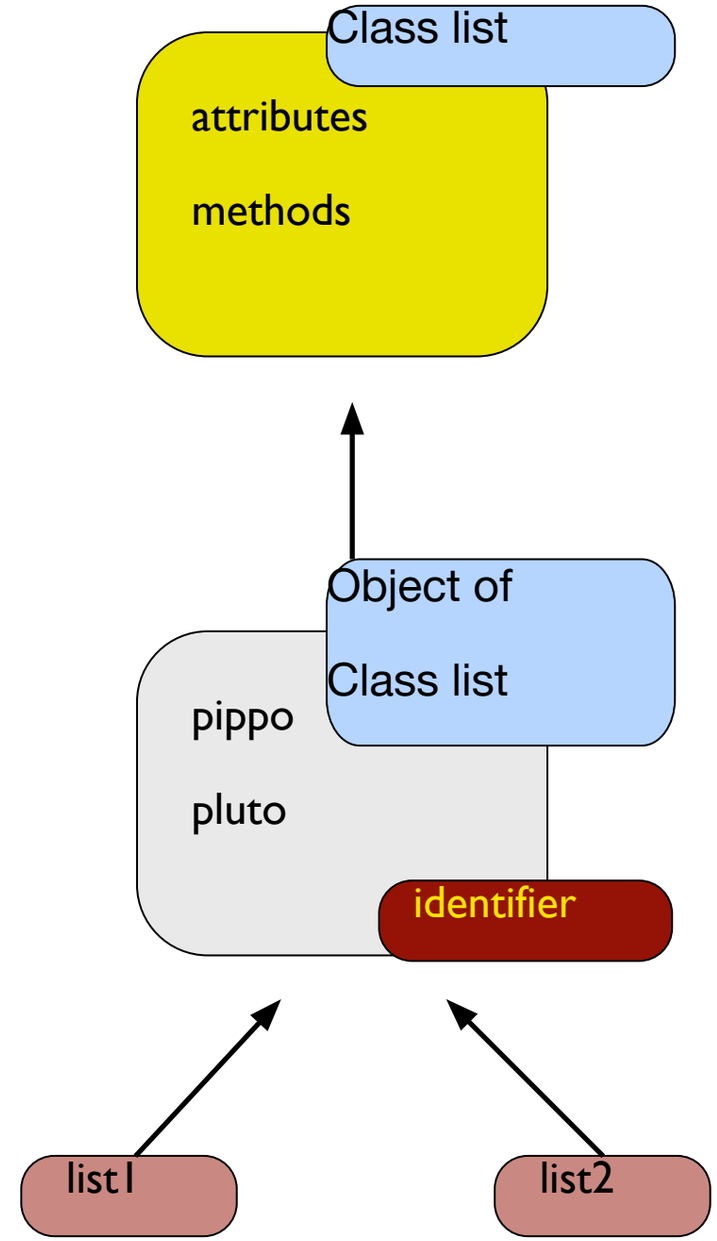
pippo and paperino

Aliasing

- You do not get a copy, but just an *alias pointing to the same list*
- You can check this with the id() function
  - The ID number of a variable identifies the object referred by that variable

```
list1 = [“pippo”, “pluto”]
list2 = list1
list2[1] = “paperino”
print(list1)
print(id(list1))
print(id(list2))
```

These are the same!

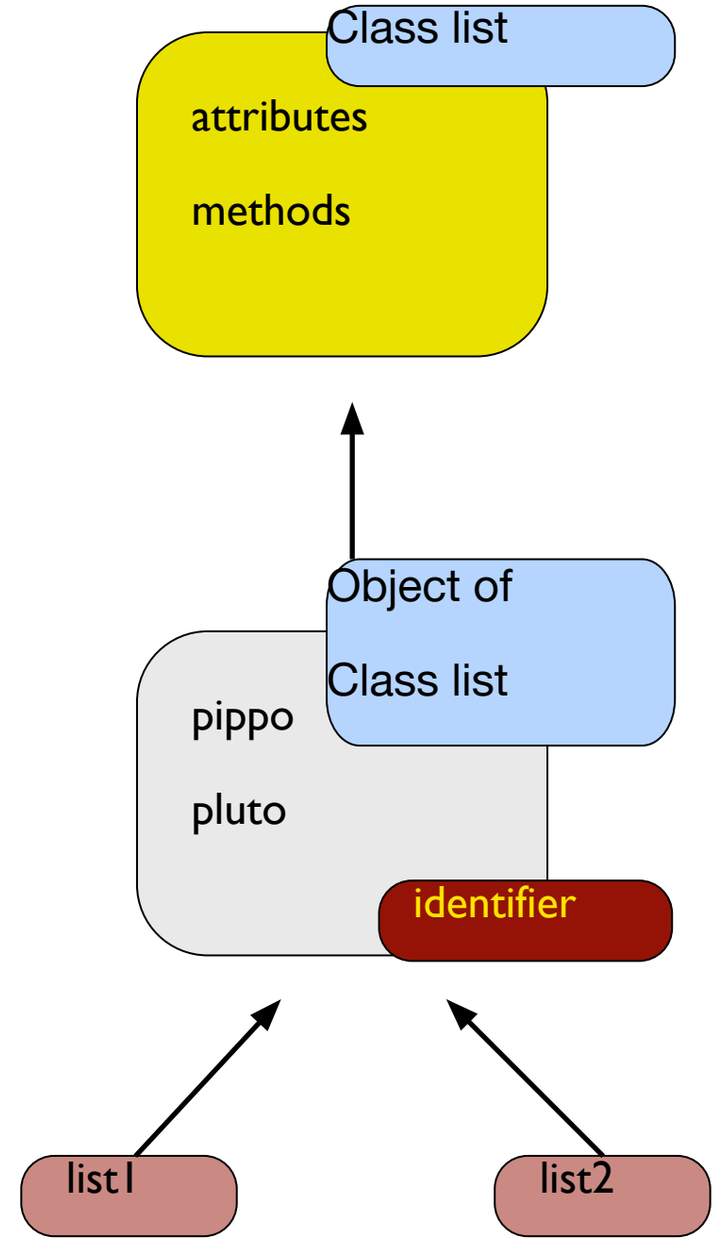


Aliasing

- You do not get a copy, but just an *alias pointing to the same list*
- You can check this with the id() function
- You can compare the ID's of two variables using is

```
list1 = ["pippo", "pluto"]
list2 = list1
list2[1] = "paperino"
print(list1 is list2)
```

Checks for aliases



Copying

- So, how can we do an actual copy of the object (by using a trick)?

```
list1 = [“pippo”, “pluto”]
list2 = ????
list2[1] = “paperino”
print(list1)
```

pippo and pluto

Copying

- You can do an actual copy of the object, by using a trick

```
list1 = [“pippo”, “pluto”]
list2 = list1[:]
list2[1] = “paperino”
print(list1)
```

pippo and pluto

Copying

- Let’s dig more

```
list1 = ["pippo", "pluto", ["qui", "quo", "qua"]]
list2 = list1[:]
list2[1] = "paperino"
print(list1)
```

Copying

- Let’s dig more

```
list1 = ["pippo", "pluto", ["qui", "quo", "qua"]]
list2 = list1[:]
list2[1] = "paperino"
print(list1)
```

No changes to list1!

Copying

- Let’s dig more

```
list1 = ["pippo", "pluto", ["qui", "quo", "qua"]]
list2 = list1[:]
list2[1] = "paperino"
print(list1)
```

list1 changed

Copying

- Copies are *shallow*
  - Elements in the original list that are lists themselves, are not copied but aliased!

```
list1 = ["pippo", "pluto", ["qui", "quo", "qua"]]
list2 = list1[:]
list2[1] = "paperino"
print(list1)
list2[2][0] = "paperone"
print(list1)
```

list1 changed!

Copying

- To create a *deep* copy, you need to use the `deepcopy()` function from the `copy` module

```
list1 = ["pippo", "pluto", ["qui", "quo", "qua"]]
list2 = deepcopy(list1)


list2[2][0] = "paperone"
print(list1)
```

No changes to list1!

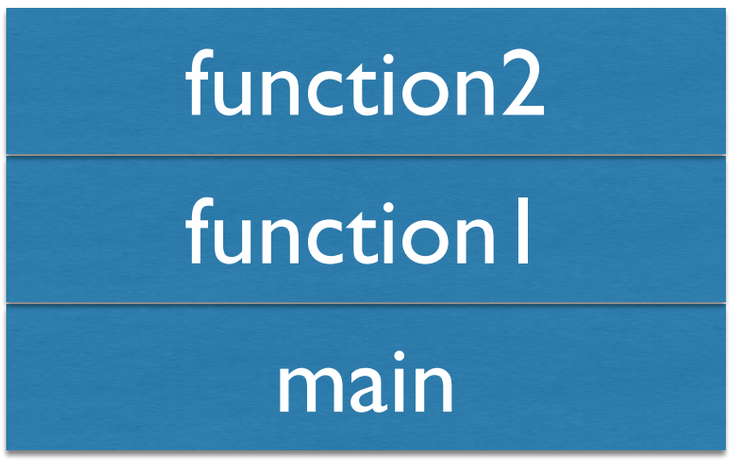
Mutable parameters

- When a mutable data structure is passed as a parameter to a function, it is **passed by reference**!
    - It can be changed by the function, and effects can be seen outside the function itself

```
def changelist(x):
    x[0] = "CHANGED!"


list1 = ["pippo", "pluto", ["qui", "quo", "qua"]]
changelist(list1)
print(list1)
```

Mutable parameters

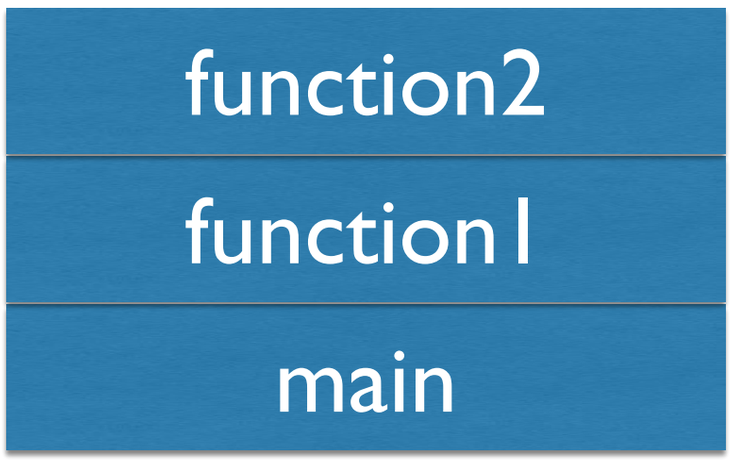
- When a mutable data structure is passed as a parameter to a function, it is **passed by reference**!
    - It can be changed by the function, and effects can be seen outside the function itself



that calls
function2
main calls
function1

Mutable parameters

- When a mutable data structure is passed as a parameter to a function, it is **passed by reference**!
    - It can be changed by the function, and effects can be seen outside the function itself



by reference: the reason for this is because lists can be quite long, and it might become hard to handle them on the stack

Other facts about lists

- Elements of a lists can be list
  - i.e., lists can be nested
- You can cast a tuple to a list using the function list(tuple)
- You can create a list using **list comprehension**
  - x*x for x in range(1,26)
    - interpreted as “x*x such that….”

As usual, all details on the BOOK!

(Proposed) Exercises (1)

- Write a program that reads an integer number n, creates a list that contains the first n positive integers, and prints it
- Write and test a function that, givem an integer x and a list, returns the position of x in the list, or -1 if x is not in the list
- Write a program that reads from input a list of n integers and displays the sum, the mean, the max and the min only of the even numbers in the list
- Write a program that reads from input a list of n integers and displays the sum and the mean of the numbers in the list that are even, divisible by 3 and greater than the previous one (assume the conditions to be true for the first element)

(Proposed) Exercises (2)

- Write a program that asks the user to enter some data, for instance the names of their friends. When the user wants to stop providing inputs, he just presses `Enter`. The program then displays an alphabetically sorted list of the data items entered. Do not just print the list, but print each item separately, on a different line.
- Sort a list of numbers using their absolute values; use the `abs()` function as key
- Count how often each letter occurs in a string (case-insensitively). You can ignore every character that is not a letter. Store the counts in a list of 26 items that all start at zero. Print the resulting counts. As index you can use `ord(letter) - ord("a")`, where letter is a lower case letter
- **All** exercises in Chapter 12 of the reference book

Programming For Data Science

Part Twelve:

Dictionaries

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Dictionaries

any immutable data type

can be a key

unordered collections of

elements

```
{'key': 'value'}
```

curly brackets to create a

dictionary

Dictionaries

- Unordered collections of elements

```
items = {“blocks”: 50, “pens”: 20}

print(items[“pens”])

for key in items:
    print(items[key])
```

use key to access an element’s value

Dictionaries

- Unordered collections of elements

```
items = {“blocks”: 50, “pens”: 20}

print(items[“pencils”])
```

runtime error!

Dictionaries

- Unordered collections of elements

```
items = {“blocks”: 50, “pens”: 20}


items[“pencils”] = 40
```

add a new element

Dictionaries

- Unordered collections of elements

```
items = {“blocks”: 50, “pens”: 20}

items[“blocks”] = 51
```

updates an element’s value

Dictionaries

- Unordered collections of elements

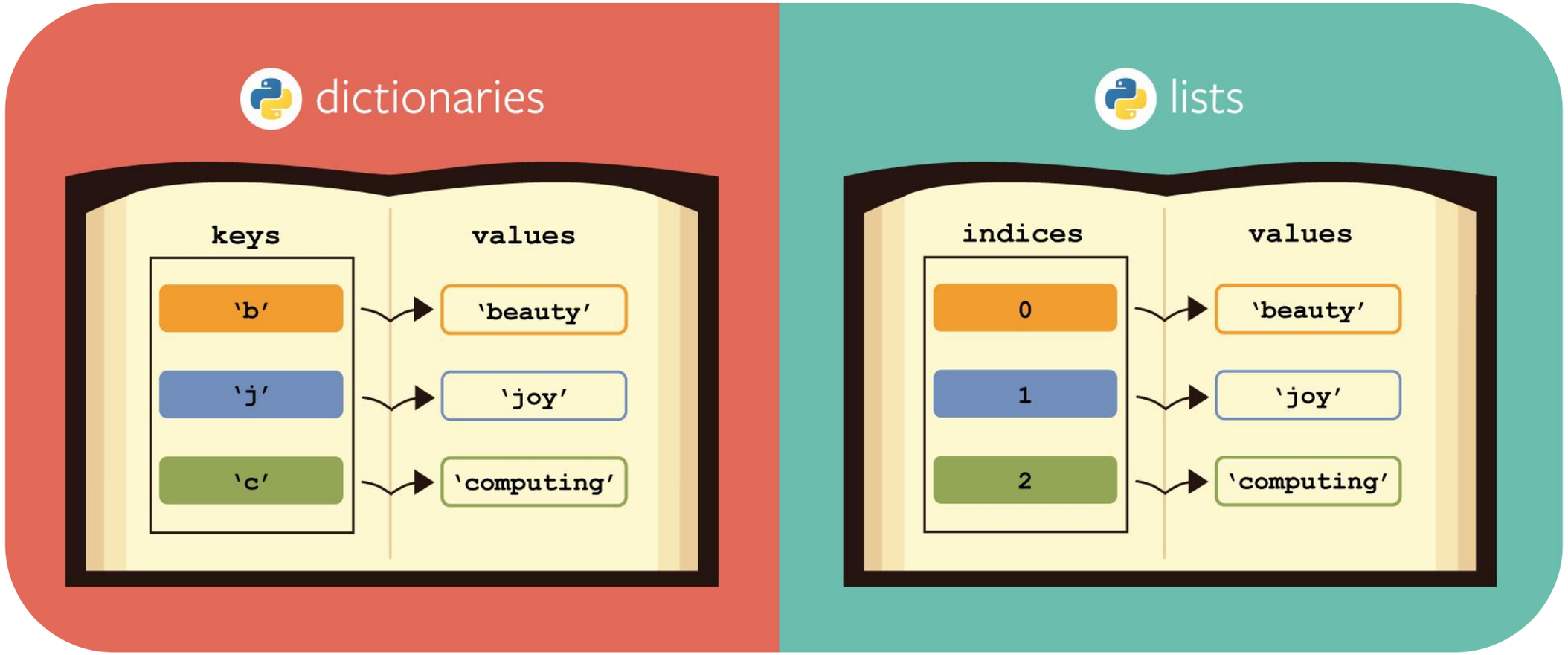
```
items = {“blocks”: 50, “pens”: 20}


del(items[“pencils”])
```

deletes an element

(**error if element does not exists**)

Dictionaries vs Lists

- **Unordered collections of elements**
  - Being unordered, you cannot sort them or reverse them
    - Different from lists



Methods

- copy()
  - Copying thru variable's assignment suffers of aliasing as for lists
  - Cannot use the 'slice-syntax' used for lists (dictionaries are unordered)
  - Use the copy() method, that makes a *shallow copy*
    - For a *deep copy*, use the copy module

```
items2 = items.copy()
```

Methods

- keys(): iterator for all keys
- values(): iterator for all values
- items(): iterator for key-values pair (as tuple)

```
for key in items.keys():
    print(key, items[key])


print (sum(items.values()))


print(items.items())
```

Methods

- get()
  - get a value, even if the key does not exists
    - Returns None if key is not present

```
pens = items.get(“pens”)

oranges = items.get(“oranges”)
```

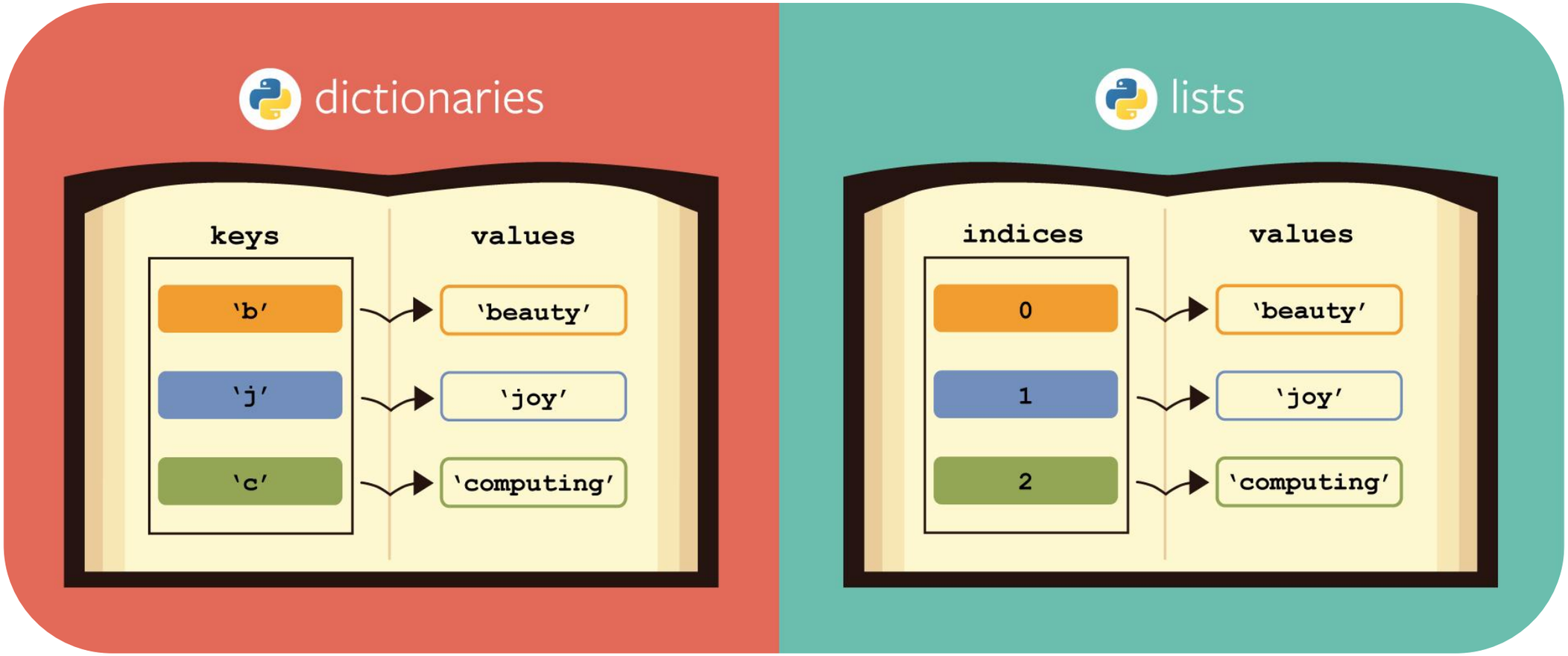
None

Keys and values

- Any immutable type can be used as key
  - int, string, float, tuples
- Values can be complex
  - Lists
  - Dictionaries

Dictionaries vs Lists

- Whether to use a dictionary or a list strongly depends on the application
- Dictionaries are implemented using *hash-tables*
- Study the example on the book in Section 13.5



(Proposed) Exercises (1)

- The code block below contains a very small dictionary that contains the translations of English words to Italian. Write and test program that uses this dictionary to create a word-for-word translation of the given sentence. A word for which you cannot find a translation, you can leave “as is.” The dictionary is supposed to be used case-insensitively, but your translation may consist of all lower case words. It is nice if you leave punctuation in the translation, but if you take it out, that is acceptable (as leaving punctuation in is quite a bit of work, and does not really have anything to do with dictionaries – besides, leaving punctuation in is much easier to do once you have learned about regular expressions).

```
en_it= { "last":"ultimo", "week":"settimana", "the":"il", "royal":"reale", "festival":"festa", "hall":"sala", "saw": "sega",
"first":"primo", "performance":"esibizione", “of":"di", "a":"un", "new":"nuovo", "symphony":"sinfonia", "by":"da",
"one":"uno", "world":"mondo", “leading”:”alla guida di”, "modern": "moderno", "composer":"compositore",
"composers":"compositori", “two":"due" }
```

(Proposed) Exercises (2)

- Write a program that reads from input n pairs name, telephone number, and inserts the, in a dictionary. Then, read a name and display its telephone number, or “not present” if the name is not in the dictionary
- Write a program that reads n integers and, for each integer, displays its frequency
- **All** exercises in Chapter 13 of the reference book

Programming For Data Science

Part Thirteen:

Sets

Giulio Rossetti
giulio.rossetti@isti.cnr.it

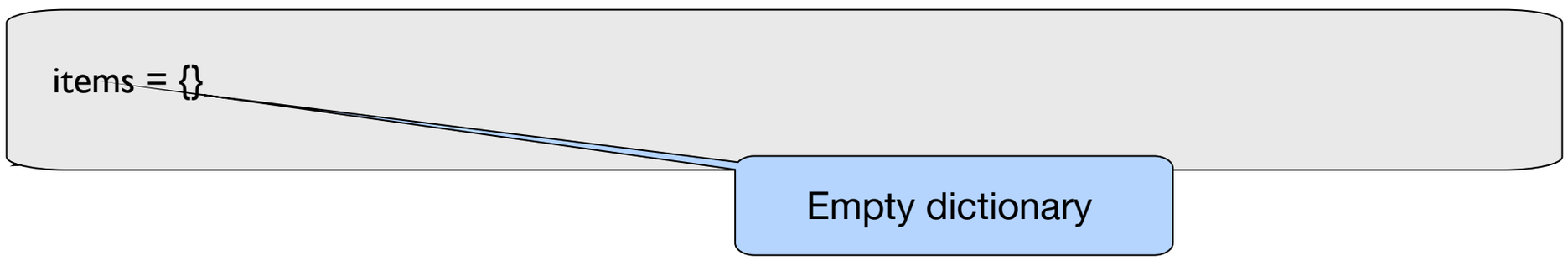
Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Sets

- Unordered collections of elements
  - Accessed using `for` loop
    - No access via index or key
  - Implemented using dictionaries, where keys are the sets' elements
    - Thus, only immutable data can be used as elements in sets
      - Sets are mutable

Sets

- Unordered collections of elements
  - Accessed using for loop
    - No access via index or key
  - Implemented using dictionaries, where keys are the sets' elements
    - Thus, only immutable data can be used as elements in sets
      - Sets are mutable



Sets

- Unordered collections of elements
  - Accessed using for loop
    - No access via index or key
  - Implemented using dictionaries, where keys are the sets' elements
    - Thus, only immutable data can be used as elements in sets
      - Sets are mutable

```
items = {}
items_set = set()
```

Empty set

Sets

- Unordered collections of elements
  - Accessed using for loop
    - No access via index or key
  - Implemented using dictionaries, where keys are the sets' elements
    - Thus, only immutable data can be used as elements in sets
      - Sets are mutable

```
items = {}
items_set = set()
items_set = {"blocks", "pens"}
```

Initialise a set

Sets

- Unordered collections of elements
  - Accessed using for loop
    - No access via index or key
  - Implemented using dictionaries, where keys are the sets' elements
    - Thus, only immutable data can be used as elements in sets
      - Sets are mutable

```
items = {}
items_set = set()
items_set = {"blocks", "pens"}
items_set2 = set("pippo")
```

Creates **a set** with the elements in the collection passed as argument

items_set2={????}

Sets

- Unordered collections of elements
  - Accessed using for loop
    - No access via index or key
  - Implemented using dictionaries, where keys are the sets' elements
    - Thus, only immutable data can be used as elements in sets
      - Sets are mutable

```
items = {}
items_set = set()
items_set = {"blocks", "pens"}
items_set2 = set("pippo")
```

Creates **a set** with the elements in the collection passed as argument

items_set2={'p', 'i', 'o'}

Methods

- add(element)
  - Adds a new element to the set
- update(list) or update(tuple)
  - Adds more than one element to the set

```
items_set = {“blocks”, “pens”}
items_set.add(“pippo”)
items_set.update([“pluto”, “paperino”])
items_set.update(“qui”)
```

Each letter is added as a new element

Methods

- add(element)
  - Adds a new element to the set
- update(list) or update(tuple)
  - Adds more than one element to the set

```
items_set = {“blocks”, “pens”}
items_set.add(“pippo”)
items_set.update([“pluto”, “paperino”])
items_set.update(“qui”)
```

How to add the word “qui” and not its letters?

Methods

- add(element)
  - Adds a new element to the set
- update(list) or update(tuple)
  - Adds more than one element to the set

```
items_set = {“blocks”, “pens”}
items_set.add(“pippo”)
items_set.update([“pluto”, “paperino”])
items_set.update(“qui”)
```

items_set.update([“qui”])

Methods

- remove(element) and discard(list)
  - Removes the element
- clear()
  - Removes all elements from the set

```
items_set = {“blocks”, “pens”}
items_set.remove(“blocks”)
items_set.discard(“blocks”)
items_set.clear()
```

Error if element is not in the set

NO Error if element is not in the set

Methods

- pop()
  - Removes an element and returns it
    - You do not know which one
- copy()
  - Like lists and dictionaries

```
items_set = {“blocks”, “pens”}
items_set.pop()


items2 = items_set.copy()
```

Methods

- union(set) or | operator
  - Union of two sets

```
items_set = {“blocks”, “pens”}
items_set2 = {“pencils”}
union = items_set.union(items_set2)


union = items_set | items_set2
```

Methods

- intersection(set) or & operator
  - Intersection of two sets
  - If empty, set() is displayed (empty set)

```
items_set = {“blocks”, “pens”}
items_set2 = {“pens”}
inters = items_set.intersection(items_set2)


inters = items_set & items_set2
```

Methods

- difference(set) or - operator
  - Difference of two sets: elements that the first set has and the second has not
  - If empty, set() is displayed (empty set)

```
items_set = {“blocks”, “pens”}
items_set2 = {“pens”}

diff = items_set.difference(items_set2)
```

Methods

- isdisjoint(set)
  - True if the two sets share no elements
- issubset(set)
  - True if all the elements of the first set are also found in the argument set
- issuperset(set)
  - True if all the elements of the argument set are also found in the first set.

Immutable sets

- frozenset()
  - The elements, once assigned, cannot be changed
    - Are immutable

```
items_set = frozenset([“blocks”, “pens”])
items_set2 = frozenset({“pens”})


union = items_set | items_set2
```

(Suggested) Exercises

- **All** exercises in Chapter 14 of the reference book