A Tutorial on the Python Programming Language

by Ricardo Aler

But first …

- Sit in pairs
- Start your computer.
- Do you have an account? Open one in here:

https://www.lab.inf.uc3m.es/servicios/apertura-de-cuenta/

What is Python?

- General-purpose, high-level programming language
- Code is very readable
- Includes different ways of programming:
  - Object-oriented
  - Imperative
  - Functional programming
- Python 2.x (2.7) vs. Python 3.x: most scientific packages

Languages for data analysis poll

2015 primary programming language:	
R (and its packages) (263)	51% (of 2015 votes)
Python (including scikit-learn and other libraries) (151)	29%
Other (Java, MATLAB, SAS, Scala, etc) (89)	17%
none (9)	1.8%
2014 primary programming language:	
R (and its packages) (237)	46% (of 2014 votes)
Python (including scikit-learn and other libraries) (117)	23%
Other (Java, MATLAB, SAS, Scala, etc) (118)	23%
none (40)	7.8%

Here is a more detailed analysis:

Python for Big Data

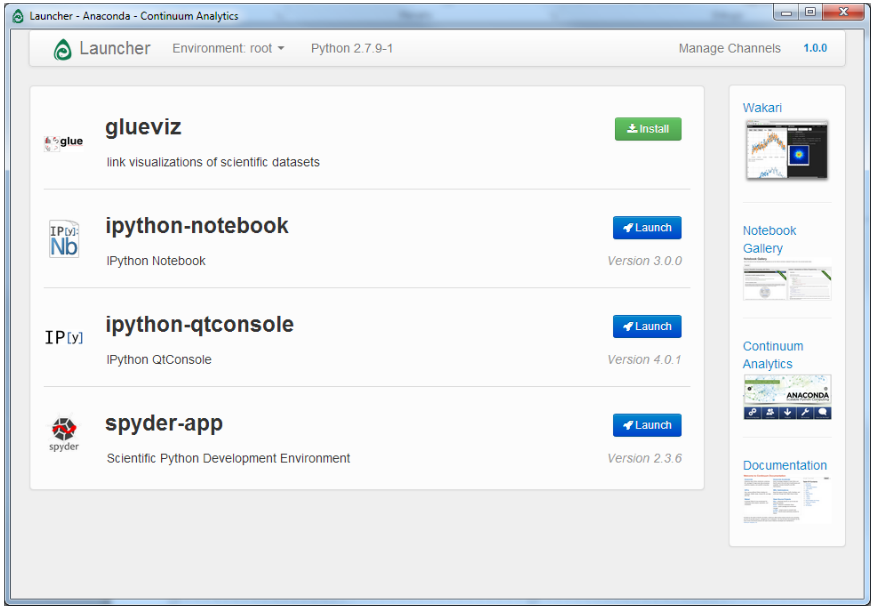
- Why Python?
- Many scientific and machine learning packages: NumPy, SciPy, scikit-learn
- Also, nice interface for Spark (pyspark)
  - R’s interface is not so well developed yet

Presentation Overview

- Anaconda (Python + machine learning packages)
- Data Types
- Control Flow
- Functions
- Files
- Modules

ANACONDA

- Free Python distribution. It includes over 300 of the most popular Python packages for science, math, engineering, data analysis.
- Launcher:
  - Ipython-qtconsole
  - Ipython-notebook
  - Spyder-app:
    - edit text files containing programs
    - + console

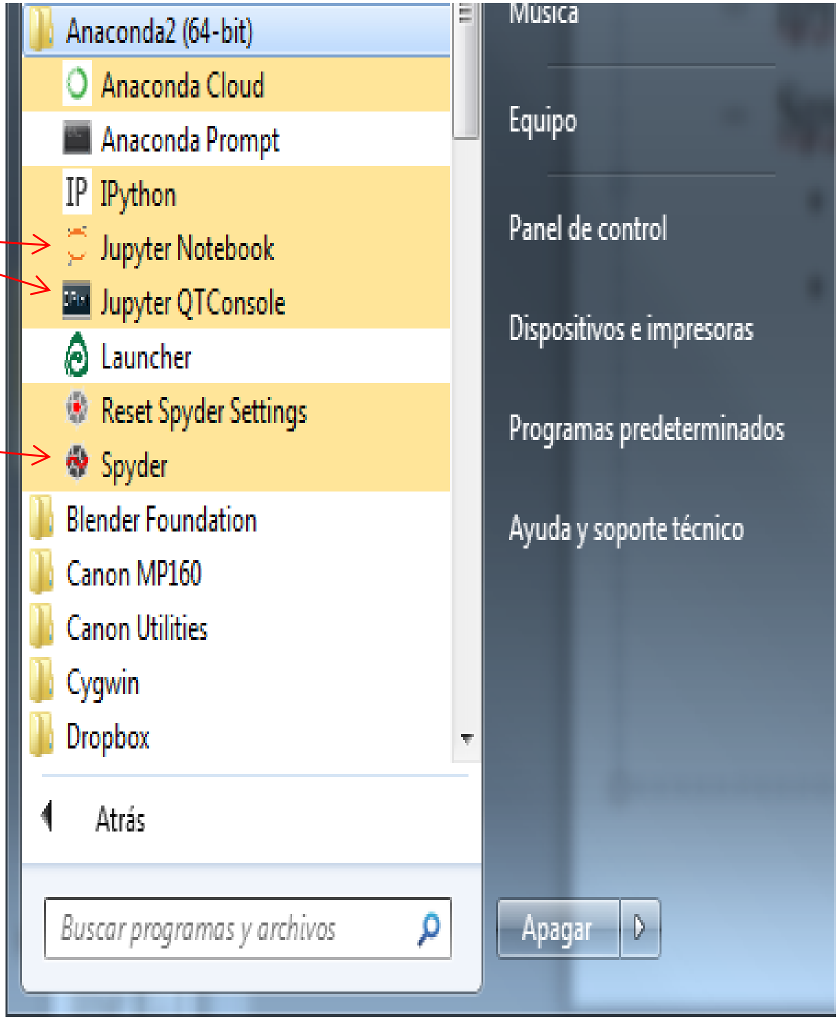


Install from: http://continuum.io/downloads

Remember to select Python 2.7!!

ANACONDA

- If Launcher does not work, start applications directly from Windows initial menu
- Ipython-qtconsole
- Ipython-notebook
  - **Jupyter**
- Spyder-app:
  - edit text files containing programs
  - + console



Interactive vs. Scripts

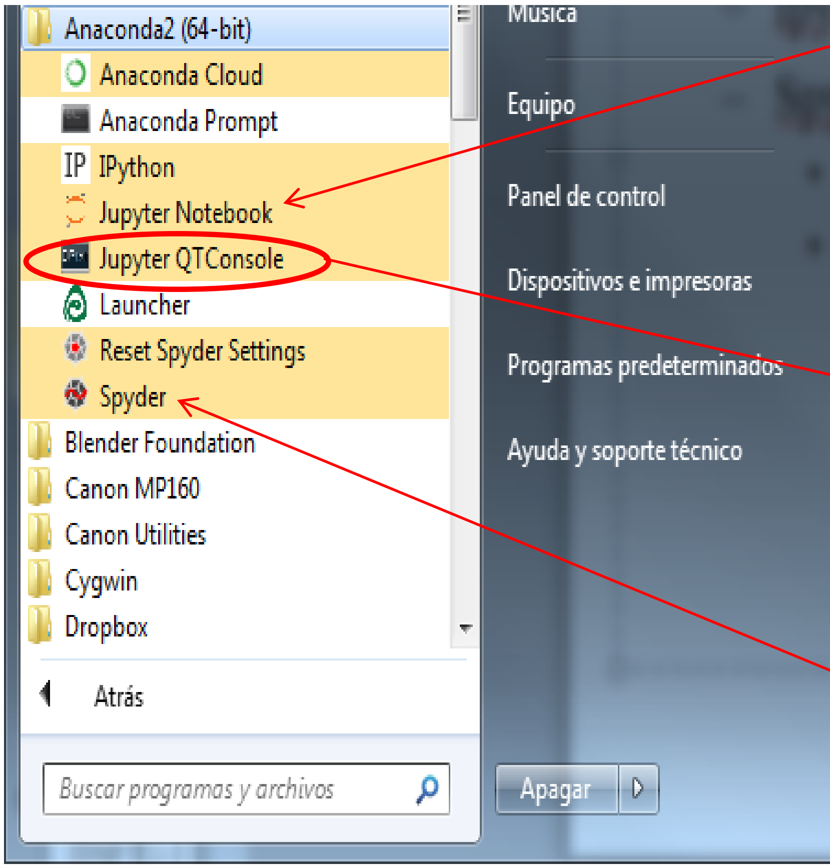
- Interactive: typing Python commands in the console (or the notebook) and obtaining an answer
- Script: a program is created using a text editor (for instance, with *spyder*)

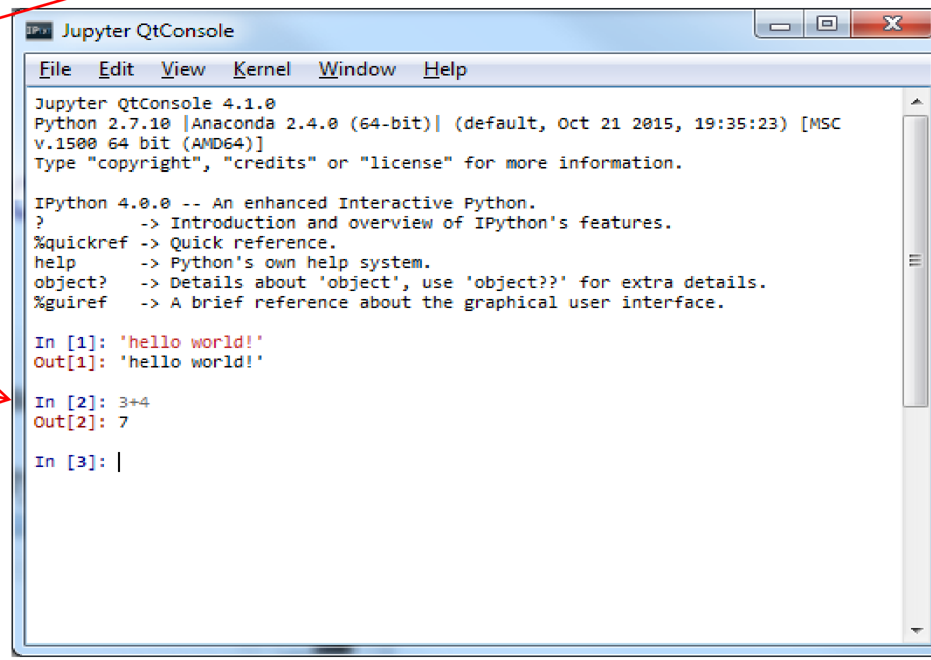
```
>>> 'hello world!'

'hello world!'
```

Interactive use: Hello World

- Open **Ipython-qtconsole** / Jupyter QTconsole
- At the prompt type ‘hello world!’

Like the qtconsole, but for the web browser





Like the qtconsole, but with a text editor
(and a whole programming environment)

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

Help

help("print")

Exercise

- Start the ipython-qtconsole
- Compute 3+4 and see the answer
- See the help of “print”

Importing Modules

- Sometimes, some functions are not directly available in Python
- They are included in modules
- Modules have to be imported in order to use its functions
- Example: '+' is included in base Python, but square root (*sqrt*). *sqrt* is included in module math

Importing Modules

If we try to use *sqrt*, we get an error:

```
In [1]: sqrt(2)
---------------------------------------------------------------------------
NameError Traceback (most recent call last)
<ipython-input-1-40e415486bd6> in <module>()
----> 1 sqrt(2)


NameError: name 'sqrt' is not defined
```

Importing Modules

Let’s import module *math*, and use the *sqrt* function within this module, by means of the dot (.) notation

```
In [2]: import math


In [3]: math.sqrt(2)
Out[3]: 1.4142135623730951
```

The print Statement

- It can be used to print results and variables
- Elements separated by commas print with a space between them
- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print 'hello'
hello
>>> print 'hello', 'there'
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Exercise

- Modules contain functions, but also constants, like pi
- Import module math, assign 2*pi to variable my_pi, and print the result

```
In [29]: import math
In [30]: math.pi
Out[30]: 3.141592653589793

In [34]: my_pi = 2*math.pi
In [35]: my_pi
Out[35]: 6.283185307179586
In [36]: print(my_pi)
6.28318530718
In [37]: print(2*math.pi)
6.28318530718
```

Variables

- The variable is created the first time you assign it a value
- Everything in Python is an object

```
>>> x = 12
>>> y = " lumberjack "
>>> x
12
>>> y
' lumberjack '
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Container: (contains other elements)
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Object types in Python with numpy module

- Container:
  - Vectors and matrices:

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

Object types in Python with Pandas module

- Container:
  - Dataframes:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Object types in Python

- Atomic: **numbers**, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Numbers

- integer: 12345, -32
- Python integer: 999999999L
- float: 1.23, 4e5, 3e-4
- octal: 012, 0456
- hex: 0xf34, 0X12FA
- complex: 3+4j, 2J, 5.0+2.5j

Operations with numbers:

- +, -, *, /
- **: power
- // integer division
- % division remainder
- …

```
>>> 123 + 222                # Integer addition
345
>>> 1.5 * 4                  # Floating-point multiplication
6.0
>>> 2 ** 100                 # 2 to the power 100
1267650600228229401496703205376
```

Object types in Python

- Atomic: numbers, **booleans (true, false),** …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Booleans

Whether an expression is true or false

- Values: True, False

Comparisons: ==, <=, >=, !=, …

```
In [18]: 3 == 3
Out[18]: True

In [19]: 3 == 4
Out[19]: False

In [20]: 3 < 4
Out[20]: True

In [21]: "aa" < "bb"
Out[21]: True
```

Combinations: and, or, not

```
In [26]: (3 == 3) and (3 < 4)
Out[26]: True

In [27]: (3 == 3) or (3 < 4)
Out[27]: True

In [28]: not((3 == 3) or (3 < 4))
Out[28]: False
```

Booleans

- Notes:
  - 0 and None are false
  - Everything else is true
  - True and False are just aliases for 1 and 0 respectively

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Container:
  - Sequences:
    - **Strings:** “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

String Literals

- They can be defined either with double quotes (“) or single quotes (‘)

```
In [30]: "Hello world"
Out[30]: 'Hello world'

In [31]: 'hello world'
Out[31]: 'hello world'
```

- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```

String Literals: multi-line

- Using triple quotes, strings can be defined across multiple lines

```
>>> s = """ I’m a string
much longer
than the others"""

>>> print s
 I’m a string
though I am much longer
than the others :)‘
```

Strings: some functions

- **len**(string) – returns the number of characters in the String
- **str**(object) – returns a String representation of the Object

```
In [56]: x = 'ABCDEF'
In [57]: len(x)
Out[57]: 6
In [58]: str(10.1)
Out[58]: '10.1'
```

Strings: some functions

- Some string functions are available only within a module, and the dot (.) notation must be used (similarly to *math.sqrt()*). The module for strings is called *str*. This module is imported automatically by the system.
- For instance, *lower()* and *upper()* are two such functions:

```
In [73]: x = 'It was the best of times, it was the worst of times‘

In [74]: str.lower(x.lower)            # Convert to lowercase
Out[74]: 'it was the best of times, it was the worst of times‘

In [75]: str.upper(x)                  # Convert to uppercase
Out[75]: 'IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES'
```

String functions

- Other string functions: *count*, *split*, *replace*

```
In [73]: x = 'It was the best of times, it was the worst of times'

In [77]: str.count(x, 'was')             # count counts how many times ‘was’ appears in x
Out[77]: 2

In [79]: print(str.split(x, ' '))         # split splits string x with space ‘ ‘ separator
['It', 'was', 'the', 'best', 'of', 'times,', 'it', 'was', 'the', 'worst', 'of', 'times']

In [80]: str.replace(x, 'was', 'is') # replace replaces ‘was’ by ‘is’ wherever it appears in x
Out[80]: 'It is the best of times, it is the worst of times'
```

String functions

IMPORTANT!!

- Typically, if you can call a function as module.function(object, other arguments), you can also use another equivalente (but shorter) syntax: object.function(other arguments)
- That is, there are two different (but equivalent) ways:
  1. object.function(arguments)
  2. module.function(object, arguments) # We already know this one
- Examples: In [32]: x = 'It was the best of times, it was the worst of times'

```
In [33]: x.lower()
Out[33]: 'it was the best of times, it was the worst of times'

In [34]: # is equivalent to

In [35]: str.lower(x)
Out[35]: 'it was the best of times, it was the worst of times'
```

```
In [36]: x.upper()
Out[36]: 'IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES'

In [37]: # is equivalent to

In [38]: str.upper(x)
Out[38]: 'IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES'
```

String functions: 2 ways

IMPORTANT!!

- That is, there are two different (but equivalent) ways:
  1. object.function(arguments)
  2. module.function(object, arguments)   # We already know this one
- Note: Use dir(‘   ‘) to see all methods for strings (dir(3) shows all methods for integers, etc.)
- Examples: In [32]: x = 'It was the best of times, it was the worst of times'

```
In [39]: x.count('was')
Out[39]: 2

In [40]: # is equivalent to

In [41]: str.count(x, 'was')
Out[41]: 2
```

```
In [45]: x.replace('was', 'is')
Out[45]: 'It is the best of times, it is the worst of times'

In [46]: # is equivalent to:

In [47]: str.replace(x, 'was', 'is')
Out[47]: 'It is the best of times, it is the worst of times
```

```
In [42]: print(x.split(' '))
['It', 'was', 'the', 'best', 'of', 'times,', 'it', 'was', 'the', 'worst', 'of', 'times']

In [43]: # is equivalent to:

In [44]: print(str.split(x, ' '))
['It', 'was', 'the', 'best', 'of', 'times,', 'it', 'was', 'the', 'worst', 'of', 'times']
```

String functions: 2 ways

IMPORTANT!!

- That is, there are two different (but equivalent) ways:
  1. object.function(arguments)
  2. module.function(object, arguments)   # We already know this one

```
In [39]: x.count('was')
Out[39]: 2

In [40]: # is equivalent to

In [41]: str.count(x, 'was')
Out[41]: 2
```

a) Notice that the first way is shorter and you don't need to remember the name of the module (*str*)

b) Only those methods listed with *dir*('was') can be used

Exercise: string functions

- Split a sentence *x* using both syntax cases:
  - First case: using *split* as a function of *x* (*x.split*)
  - Second case: using *split* as a function of module *str* (*str.split(x)*)

```
In [12]: x = 'It was the best of times, it was the worst of times'
In [13]: x
Out[13]: 'It was the best of times, it was the worst of times'
```

```
In [14]: # First case
In [15]: x.split(' ')
Out[15]:
['It',
'was',
'the',
'best',
'of',
'times,',
'it',
'was',
'the',
'worst',
'of',
'times']
```

```
In [16]: # Second case: split as function of module str
In [17]: str.split(x, ' ')
Out[17]:
['It',
'was',
'the',
'best',
'of',
'times,',
'it',
'was',
'the',
'worst',
'of',
'times']
```

Positive indices	0	1	2	3	4	5
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’

Substrings (slicing)

Slicing = obtaining substrings from strings

```
>>> s = '012345'
>>> s[0]
‘0’
>>> s[1]
‘1’
>>> s[3]
'3'
>>> s[1:4]
'123'
```

- Generic slicing sentence: s[start:end:by]
  - Obtain elements from *start* to *(end-1)* with steps of “*by*”
- IMPORTANT*:*
  - *start* begins at 0!!
  - The slice (or substring) includes values from *start* to *end-1*!!!
- *start* >= 0
- *end* < *len(s)*
- *by*: step

Positive indices	0	1	2	3	4	5
Negative indices	-6	-5	-4	-3	-2	-1
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’

Substrings (slicing)

```
>>> s = '012345'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-1]
‘5‘
>>> s[-2]
‘4’
>>> s[-6]
‘0’
```

Generic sentence: s[start:end:by]

Excluding *start* or *end* is the same as index 0 or last index, respectively

s[2:] == s[2:6] == s[2:len(s)]
s[:4] == s[0:4]

Negative indices start at the end of the string

s[-1] == s[5] == s[len(s)-1]
s[-2] == s[4] == s[len(s)-2]
s[-6] = s[-len(s)] == s[0]

Substrings (slicing)

Slicing = obtaining sublists from strings (or from lists)

Positive indices	**0**	**1**	**2**	**3**	**4**	**5**
Negative indices	-6	-5	-4	-3	-2	-1
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’
string2	‘A’	‘B’	‘C’	‘D’	‘E’	‘F’

```
>>> string2 = ‘ABCDEF'
>>> string2[2:]
‘CDEF'
>>> s[:4]
‘ABCDE'
```

```
>>> string2[-1]
‘F‘
>>> string2[-2]
‘E’
>>> string2[-6]
‘A’
```

Substrings (slicing)

- Generic sentence: s[start:end:**by**]
- by: step

```
>>> s = '012345'
>>> s[0:4:2]
'02'
>>> s[0::2]
'024'
>>> s[-1::-1]
'543210'
>>> s[-1::-2]
'531'
```

s[0:4:2]: Get indices from 0 to 3 by 2 (even indices)

s[0::2]: Get indices from 0 to end by 2 (even indices)

s[-1::-1]: Get indices from end to beginning by -1 (reverse order)

s[-1::-2]: Get indices from end to beginning by -2 (indices 5, 3, 1 (or equivalently -1, -3, -5)

Exercise

1. Create any string, for instance:

   ‘In a village of La Mancha, the name of which I have no desire to call to mind’

2. Convert it to uppercase:

   'IN A VILLAGE OF LA MANCHA, THE NAME OF WHICH I HAVE NO DESIRE TO CALL TO MIND'

3. Obtain another string by keeping one character every four characters (via slicing):

   'I L LAAHAOH ANEE L D'

Exercise: solution

```
In [76]: x = 'In a village of La Mancha, the name of which I have no desire to call to mind'

In [77]: x = x.upper()

In [78]: x
Out[78]: 'IN A VILLAGE OF LA MANCHA, THE NAME OF WHICH I HAVE NO DESIRE TO CALL TO MIND'

In [79]: y = x[0::4]

In [80]: y
Out[80]: 'I L LAAHAOH ANEE L
```

String Formatting (1): %

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

String Formatting (2): format

- <formatted string>.format(<elements to insert>)

```
>>> "One, {}, three".format(2)
'One, 2, three'
>>> “{}, two, {}".format(1,3)
'1, two, 3'
>>> "{} two {}".format(1, 'three')
'1 two three'
>>> "{0} two {1}".format(1, 'three')
'1 two three‘
>>> "{1} two {0}".format(1, 'three')
'three two 1'
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - **Lists:** [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Lists

- Ordered collection of data
- Elements can be of different types
- Same subset (slicing) operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

Lists are *mutable* (i.e. they can be modified. Strings cannot)

- **x[i] = a** reassigns the ith element to the value a
- Important: variables contain references (pointers) to the object, not the object itself
- Since *x* and *y* point to the same list object, *both* are changed

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
```

Lists: references vs. copies

- If a copy is needed instead of a reference, the copy function can be used (import copy)

Reference: x and y are the same thing

```
In [58]: x = [1, 2, 3]
In [59]: y = x

In [60]: x[1] = 15

In [61]: x
Out[61]: [1, 15, 3]
In [62]: y
Out[62]: [1, 15, 3]
```

Copy: a and b are different things

```
In [63]: import copy
In [64]: a = [1, 2, 3]
In [65]: b = copy.deepcopy(a)

In [66]: a[1] = 15

In [67]: a
Out[67]: [1, 15, 3]
In [68]: b
Out[68]: [1, 2, 3]
```

Exercise: lists modifying content

1. Create a variable called *list* with numbers 1, 10, 100, 1000, 10000, 1000000
2. Modify variable *list* via slicing so that 0 appears instead of 1000

Exercise: solution

1. Create a variable called *list* with numbers 1, 10, 100, 1000, 10000, 1000000
2. Modify variable *list* via slicing so that 0 appears instead of 1000

In [**115**]: x = [1,10,100,1000,10000, 1000000]
In [**116**]: x[3] = 0
In [**117**]: x
Out[**117**]: [1, 10, 100, 0, 10000, 1000000]

Lists: Modifying Content

Lists are *mutable* (i.e. they can be modified)

- **x[i:j:k] = b** reassigns the sublist defined by *i:j:k* to list *b*

In [**7**]: x = [0, 1, 2, 3, 4, 5]
In [**8**]: y = x
In [**9**]: x[1:3] = ['one', 'two', 'three']
In [**10**]: x
Out[**10**]: [0, 'one', 'two', 'three', 3, 4, 5]
In [**11**]: y
Out[**11**]: [0, 'one', 'two', 'three', 3, 4, 5]

Lists: Modifying Content

- **x.append(12)** inserts element *12* at the end of the list
- **x.extend([13, 14])** extends list [12, 13] at the end of the list
- In both cases the original list is modified!!!

```
In [14]: x = [1,2,3]
In [15]: x.append(12)
In [16]: x
Out[16]: [1, 2, 3, 12]
In [18]: x.extend([13, 14])
In [19]: x
Out[19]: [1, 2, 3, 12, 13, 14]
```

- + also concatenates lists, but it does not modify the original list

```
In [20]: y = [1, 2, 3]
In [21]: y + [13, 14]
Out[21]: [1, 2, 3, 13, 14]
In [22]: y
Out[22]: [1, 2, 3]
```

Reminder: two ways of calling functions on objects

- Let us remember that there are two ways of applying functions to lists (just as with strings):
  1. module.function(object, …)
  2. object.method(…)

```
In [27]: x = [1, 2, 3]
In [28]: list.extend(x, [13, 14])
In [29]: x
Out[29]: [1, 2, 3, 13, 14]

# is equivalent to:

In [30]: x = [1, 2, 3]
In [31]: x.extend([13, 14])
In [32]: x
Out[32]: [1, 2, 3, 13, 14]
```

Lists: deleting elements

- Function *del*:

```
In [33]: x = range(10)
In [34]: x
Out[34]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [35]: del(x[1])
In [36]: x
Out[36]: [0, 2, 3, 4, 5, 6, 7, 8, 9]
In [37]: del(x[2:4])
In [38]: x
Out[38]: [0, 2, 5, 6, 7, 8, 9]
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - **Tuples:** (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - **Dictionaries:** {“R”: 51, “Python”: 29}

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*
- Example number of bottles of different drinks
- Access and modification by key

```
In [47]: d = {'milk': 3, 'beer': 21, 'olive oil': 2}
In [48]: d
Out[48]: {'beer': 21, 'milk': 3, 'olive oil': 2}
In [49]: d['milk']
Out[49]: 3
In [50]: d['milk'] = 4
In [51]: d
Out[51]: {'beer': 21, 'milk': 4, 'olive oil': 2}
```

Dictionaries: Add/Delete

- Assigning to a key that does not exist adds an entry:

```
In [52]: d['coffee'] = 3
In [53]: d
Out[53]: {'beer': 21, 'coffee': 3, 'milk': 4, 'olive oil': 2}
```

- Elements can be deleted with *del* (like with lists)

```
In [54]: del(d['beer'])
In [55]: d
Out[55]: {'coffee': 3, 'milk': 4, 'olive oil': 2}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

- Lists, Tuples, and Dictionaries are containers that can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references, but copies can be made