

Python Basics (1/2)

Computing Methods for Experimental Physics and Data Analysis

L. Baldini

luca.baldini@pi.infn.it

Università and INFN–Pisa

Compiled on September 20, 2021



Python 3 vs. Python 2

- ▷ You might have heard about the Python 2 vs. Python 3 diatribe
- ▷ What's new in Python 3?
 - ▷ `print()` is a function (vs. a statement)
 - ▷ Integer division returns a float (hurray!)
 - ▷ But really, **the big change is the implementation of unicode support**
 - ▷ Although you probably don't care that much if you are a data scientist
- ▷ Python 2 is dead, and Python 3 is the present!
 - ▷ Python 2 EOL (End Of Life) was January 1, 2020
 - ▷ Hundreds of features introduced in Python 3 and not backported to Python 2
 - ▷ All third-party modules worth mentioning support both Python 2 and Python 3, these days
- ▷ **One-line summary: we shall be using Python 3**
 - ▷ And you should, too—make sure there's no leftover Python 2 installation on your PC



The zen of Python

See PEP 20, <https://www.python.org/dev/peps/pep-0020/>

```
1  [lbaldini@nbbaldini slides]$ python
2  Python 3.7.4 (default, Jul  9 2019, 16:32:37)
3  [GCC 9.1.1 20190503 (Red Hat 9.1.1-1)] on linux
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>> import this
6  The Zen of Python, by Tim Peters
7
8  Beautiful is better than ugly.
9  Explicit is better than implicit.
10 Simple is better than complex.
11 Complex is better than complicated.
12 Flat is better than nested.
13 Sparse is better than dense.
14 Readability counts.
15 Special cases aren't special enough to break the rules.
16 Although practicality beats purity.
17 Errors should never pass silently.
18 Unless explicitly silenced.
19 In the face of ambiguity, refuse the temptation to guess.
20 There should be one-- and preferably only one --obvious way to do it.
21 Although that way may not be obvious at first unless you're Dutch.
22 Now is better than never.
23 Although never is often better than *right* now.
24 If the implementation is hard to explain, it's a bad idea.
25 If the implementation is easy to explain, it may be a good idea.
26 Namespaces are one honking great idea -- let's do more of those!
```



Coding conventions?

<https://www.python.org/dev/peps/pep-0008/>

- ▷ *Coding conventions* are guidelines about how to write code
 - ▷ Different for different languages
 - ▷ i.e., you are encouraged to stick to them, but your code will happily run if you don't
- ▷ **Then why should I care?**
- ▷ Code is read much more often than it is written
 - ▷ Readability counts (the zen of Python)
- ▷ One-line summary: **think about it but don't be obsessed by it**
- ▷ The bible of coding conventions for Python is
<https://www.python.org/dev/peps/pep-0008/>
- ▷ There are automatic tools out there to help you
 - ▷ <https://www.pylint.org/>
 - ▷ <https://pypi.org/project/pyflakes/>
 - ▷ <http://mypy-lang.org/>
 - ▷ <https://github.com/PyCQA/pycodestyle>



Variables and basic types

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/basic_types.py

```
1 i = 3
2 x = 3.0
3 print(i, type(i))
4 print(x, type(x))
5
6 s = 'Hi there!'
7 print(s, type(s))
8
9 l = [1, 2, 'a string']
10 print(l, type(l), l[0])
11
12 t = (1, 2, 'a string')
13 print(t, type(t), t[0])
14
15 d = {'key1': 1, 'key2': 2}
16 print(d, type(d), d['key1'])
17
18 [Output]
19 3 <class 'int'>
20 3.0 <class 'float'>
21 Hi there! <class 'str'>
22 [1, 2, 'a string'] <class 'list'> 1
23 (1, 2, 'a string') <class 'tuple'> 1
24 'key1': 1, 'key2': 2 <class 'dict'> 1
```



Digression: string formatting

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/string_formatting.py

```
1 name = 'Luca'
2 age = 42
3
4 # The ugly way.
5 print('My name is ' + name + ' I am ' + str(age) + ' year(s) old.')
6
7 # The old way (% operator)
8 print('My name is %s I am %d year(s) old.' % (name, age))
9
10 # The new way (.format)
11 # This is actually *much* more powerful and flexible than implied here.
12 print('My name is {} I am {} year(s) old.'.format(name, age))
13
14 # The newer way---new in Python 3.6. This is awesome!
15 print(f'My name is {name} I am {age} year(s) old.')
16
17 [Output]
18 My name is Luca I am 42 year(s) old.
19 My name is Luca I am 42 year(s) old.
20 My name is Luca I am 42 year(s) old.
21 My name is Luca I am 42 year(s) old.
```

▷ String formatting in a nutshell:

- ▷ Never add strings
- ▷ Try and avoid using the % operator
- ▷ format() is ok, although a little bit on the verbose side
- ▷ Use f-strings whenever you can (need Python 3.6+)



Defining functions

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

▷ DRY (Don't Repeat Yourself) is better than WET (Write Every Time)

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/funcl.py>

```
1 import math
2
3 def square(x):
4     """Return the square of x.
5     """
6     return x * x
7
8 def cartesian_to_polar(x=1., y=1.):
9     """Convert cartesian to polar coordinates.
10    """
11    r = math.sqrt(x**2. + y**2.)
12    phi = math.atan2(y, x)
13    return r, phi
14
15 print(square(2.))
16 print(cartesian_to_polar(0., 1.))
17 print(cartesian_to_polar())
18
19 [Output]
20 4.0
21 (1.0, 1.5707963267948966)
22 (1.4142135623730951, 0.7853981633974483)
```



Variadic functions

- ▷ Variadic functions accept a variable number of arguments
 - ▷ More elegant than passing a list or a tuple of arguments
 - ▷ How the heck is *that* implemented?

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/func_variadic1.py

```
1  import os
2
3  p1 = os.path.join('path', 'to', 'my', 'file')
4  p2 = os.path.join('howdy', 'partner')
5
6  print(p1)
7  print(p2)
8
9  s1 = sum([1, 2])
10 s2 = sum([1, 2, 3, 4, 5])
11
12 print(s1)
13 print(s2)
14
15 [Output]
16 path/to/my/file
17 howdy/partner
18 3
19 15
```




Arbitrary argument lists

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/func_variadic2.py

```
1  import os
2
3  def join1(*args):
4      """Horrible: do not use the + operator with strings in a loop.
5      """
6      out = ''
7      for arg in args:
8          out += '%s/' % arg
9      return out.rstrip('/')
10
11 def join2(*args):
12     """This a more sensible version---and you get the idea of the *.
13     """
14     return '/'.join(args)
15
16 def join3(*args, sep=os.path.sep):
17     """Even better---this will work on any OS.
18     """
19     return sep.join(args)
20
21 print(join1('path', 'to', 'file'))
22 print(join2('path', 'to', 'file'))
23 print(join3('path', 'to', 'file'))
24
25 [Output]
26 path/to/file
27 path/to/file
28 path/to/file
```



A real life example

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/func_variadic_fit.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 x = np.linspace(0., 10., 11)
6 y = 2.5 + 3.2 * x
7
8 def model(x, m, q):
9     return m * x + q
10
11 popt, pcov = curve_fit(model, x, y)
12
13 plt.errorbar(x, y, fmt='o')
14 # Overlay the model without unpacking the best-fit parameters.
15 plt.plot(x, model(x, *popt))
16
17 # Compare with
18 # mhat, qhat = popt
19 # plt.plot(x, model(x, mhat, qhat))
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/func_kwargs.py

```
1 def func(**kwargs):
2     """
3     """
4     print(kwargs.get('verbose', False))
5
6
7 func()
8 func(verbose=True)
9 func(verbose=False)
10 func(verbose=True, num_events=3)
11 func(True)
12
13 [Output]
14 False
15 True
16 False
17 True
18 Traceback (most recent call last):
19   File "snippets/func_kwargs.py", line 11, in <module>
20     func(True)
21 TypeError: func() takes 0 positional arguments but 1 was given
```



Basic control flow

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/control_flow.py

```
1 i = 2
2
3 # Conditional expressions
4 if i == 2:
5     print('Apple')
6 elif i == 3:
7     print('Peach')
8 else:
9     print('Cheese')
10
11 # For loops
12 for i in [1, 2, 3]:
13     print(i)
14
15 # While loops
16 while i != 0:
17     print(i)
18     i -= 1
19
20 [Output]
21 Apple
22 1
23 2
24 3
25 3
26 2
27 1
```



Advanced iteration

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/iteration.py>

```
1 list1 = ['a', 'b', 'c']
2 list2 = [10, 11, 12]
3
4 # Horrible (and very un-Pythonic, too!)
5 for i in range(len(list1)):
6     print(i, list1[i])
7
8 # Nice-looking.
9 for i, item in enumerate(list1):
10    print(i, item)
11
12 # Zipping iterables
13 for item1, item2 in zip(list1, list2):
14    print(item1, item2)
15
16 # List comprehension
17 print([x**2 for x in list2])
18
19 [Output]
20 0 a
21 1 b
22 2 c
23 0 a
24 1 b
25 2 c
26 a 10
27 b 11
28 c 12
29 [100, 121, 144]
```



Challenge of the day

```
1 [lbaldini@nbbaldini slides]$ python
2 Python 3.7.4 (default, Jul  9 2019, 16:32:37)
3 [GCC 9.1.1 20190503 (Red Hat 9.1.1-1)] on linux
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> 0.1 + 0.2 == 0.3
6 False
7 >>> 0.2 + 0.2 == 0.4
8 True
```

▷ What the hell?

Floating point representation

IEEE 754 standard

IEEE Floating Point Representation

s	exponent	mantissa
1 bit	8 bits	23 bits

IEEE Double Precision Floating Point Representation

s	exponent	mantissa
1 bit	11 bits	52 bits

- ▷ Floating-point number representation from left to right
 - ▷ **sign** (s , 1 bit, $0 \rightarrow -, 1 \rightarrow +$)
 - ▷ **exponent** (e , 8 or 11 bit)
 - ▷ **significand** or **mantissa** (m , 23 or 52 bit)
- ▷ The exponent does not have a sign
 - ▷ An exponent bias b is subtracted from it (127 or 1023)
- ▷ The significand MSB is assumed to be 1, unless the exponent is 0

$$x = s \times m \times 2^{e-b} \quad (1)$$



A simple example

<https://babbage.cs.qc.cuny.edu/IEEE-754/index.xhtml>

- ▷ Take a floating-point number with an exact binary representation

$$0.75_{10} = 0.11_2 = 0 \times 2 + 1 \times 2^{-1} + 1 \times 2^{-2} = \frac{1}{2} + \frac{1}{4} = 1.5 \times 2^{-1} \quad (2)$$

```
1 0.75 -> 0x3F400000 = 0b|0|01111110|100000000000000000000000
2 sign = 0b0 = 0 -> +
3 exponent = 0b01111110 = 126 -> 126 - 127 = -1
4 significand = 0b(1)100000000000000000000000 = 12582912 -> 0b1.1 = 1.5
```

- ▷ The representation of any floating point number is equivalent to the ratio of two integers, where the denominator is a power of 2

$$x = \frac{m}{2^{23-e}} = \frac{12582912}{2^{24}} = \frac{3}{4} = 0.75 \quad (3)$$



Floating point representation in Python

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/float.py>

```
1 a = 0.75
2 num, den = a.as_integer_ratio()
3 print(num, den)
4 print('{:.30f}'.format(num / den))
5
6 print()
7
8 b = 0.1
9 num, den = b.as_integer_ratio()
10 print(num, den)
11 print('{:.30f}'.format(num / den))
12
13 [Output]
14 3 4
15 0.750000000000000000000000000000
16
17 3602879701896397 36028797018963968
18 0.100000000000000005551115123126
```

- ▷ `as_integer_ratio()` returns the internal representation of a float
- ▷ Mind this is not guaranteed to be the closest rational approximation



Floating point representation

IEEE 754 standard

- ▷ Good properties:
 - ▷ numbers at wildly different magnitudes
 - ▷ same relative accuracy at all magnitudes
 - ▷ allow calculations across magnitudes
- ▷ Dynamic range dictated by the number n_e of bits in the exponent
Range: $2^{2^{n_e}-1}$
- ▷ Precision dictated by the number n_s of bits in the significand
Precision: $\log_{10}(2^{n_s+1})$

Precision	Bits	Dynamic range	Digits of precision
Single	$1 + 8 + 23 = 32$	$\approx 2^{128}$ or 10^{38}	7
Double	$1 + 11 + 52 = 64$	$\approx 2^{1024}$ or 10^{308}	15



References

- ▷ <https://scipy-lectures.org/>
- ▷ <https://docs.quantifiedcode.com/python-anti-patterns/>
- ▷ https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
- ▷ <https://www.python.org/dev/peps/pep-0020/>
- ▷ <https://www.python.org/dev/peps/pep-0008/>
- ▷ <https://docs.python-guide.org/writing/style/>
- ▷ <https://docs.python.org/3/library/stdtypes.html>
- ▷ <https://docs.python.org/3/tutorial/controlflow.html#defining-functions>
- ▷ <https://docs.python.org/3/tutorial/floatingpoint.html>
- ▷ <https://floating-point-gui.de/>
- ▷ https://www.itu.dk/~sestoft/bachelor/IEEE754_article.pdf