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 27, 2020

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 3 is the future!
  - 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

The zen of Python

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

```
[lbaldini@nbbaldini slides]$ python
Python 3.7.4 (default, Jul  9 2019, 16:32:37)
[GCC 9.1.1 20190503 (Red Hat 9.1.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

- *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

```
i = 3
x = 3.0
print(i, type(i))
print(x, type(x))

s = 'Hi there!'
print(s, type(s))

l = [1, 2, 'a string']
print(l, type(l), l[0])

t = (1, 2, 'a string')
print(t, type(t), t[0])

d = {'key1': 1, 'key2': 2}
print(d, type(d), d['key1'])

[Output]
3 <class 'int'>
3.0 <class 'float'>
Hi there! <class 'str'>
[1, 2, 'a string'] <class 'list'> 1
(1, 2, 'a string') <class 'tuple'> 1
'key1': 1, 'key2': 2 <class 'dict'> 1
```

Digression: string formatting

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

```
name = 'Luca'
age = 42

# The ugly way.
print('My name is ' + name + ' I am ' + str(age) + ' year(s) old.')

# The old way (% operator)
print('My name is %s I am %d year(s) old.' % (name, age))

# The new way (.format)
# This is actually *much* more powerful and flexible than implied here.
print('My name is {} I am {} year(s) old.'.format(name, age))

# The newer way---new in Python 3.6. This is awesome!
print(f'My name is {name} I am {age} year(s) old.')

[Output]
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
```

- String formatting in a nutshell:
  - Never add strings
  - Try and avoid using the `%` operator
  - Go for format(), and use f-strings if you can

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/func1.py

```python
import math

def square(x):
    """Return the square of x.
    """
    return x * x

def cartesian_to_polar(x=1., y=1.):
    """Convert cartesian to polar coordinates.
    """
    r = math.sqrt(x**2. + y**2.)
    phi = math.atan2(y, x)
    return r, phi

print(square(2.))
print(cartesian_to_polar(0., 1.))
print(cartesian_to_polar())

[Output]
4.0
(1.0, 1.5707963267948966)
(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

```
import os

p1 = os.path.join('path', 'to', 'my', 'file')
p2 = os.path.join('howdy', 'partner')

print(p1)
print(p2)

s1 = sum([1, 2])
s2 = sum([1, 2, 3, 4, 5])

print(s1)
print(s2)

[Output]
path/to/my/file
howdy/partner
3
15
```

Arbitrary argument lists

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

```python
import os

def join1(*args):
    """Horrible: do not use the + operator with strings in a loop.
    """
    out = ''
    for arg in args:
        out += '%s/' % arg
    return out.rstrip('/')

def join2(*args):
    """This a more sensible version---and you get the idea of the *.
    """
    return '/'.join(args)

def join3(*args, sep=os.path.sep):
    """Even better---this will work on any OS.
    """
    return sep.join(args)

print(join1('path', 'to', 'file'))
print(join2('path', 'to', 'file'))
print(join3('path', 'to', 'file'))

[Output]
path/to/file
path/to/file
path/to/file
```

A real life example

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

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

x = np.linspace(0., 10., 11)
y = 2.5 + 3.2 * x

def model(x, m, q):
    return m * x + q

popt, pcov = curve_fit(model, x, y)

plt.errorbar(x, y, fmt='o')
# Overlay the model without unpacking the best-fit parameters.
plt.plot(x, model(x, *popt))

# Compare with
# mhat, qhat = popt
# plt.plot(x, model(x, mhat, qhat))
```

Keyword arguments

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

```
def func(**kwargs):
    """
    """
    print(kwargs.get('verbose', False))


func()
func(verbose=True)
func(verbose=False)
func(verbose=True, num_events=3)
func(True)

[Output]
False
True
False
True
Traceback (most recent call last):
  File "snippets/func_kwargs.py", line 11, in <module>
    func(True)
TypeError: func() takes 0 positional arguments but 1 was given
```

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

```python
i = 2

# Conditional expressions
if i == 2:
    print('Apple')
elif i == 3:
    print('Peach')
else:
    print('Cheese')

# For loops
for i in [1, 2, 3]:
    print(i)

# While loops
while i != 0:
    print(i)
    i -= 1

[Output]
Apple
1
2
3
3
2
1
```

Advanced iteration

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

```
list1 = ['a', 'b', 'c']
list2 = [10, 11, 12]

# Horrible (and very un-Pythonic, too)!
for i in range(len(list1)):
    print(i, list1[i])

# Nice-looking.
for i, item in enumerate(list1):
    print(i, item)

# Zipping iterables
for item1, item2 in zip(list1, list2):
    print(item1, item2)

# List comprehension
print([x**2 for x in list2])

[Output]
0 a
1 b
2 c
0 a
1 b
2 c
a 10
b 11
c 12
[100, 121, 144]
```

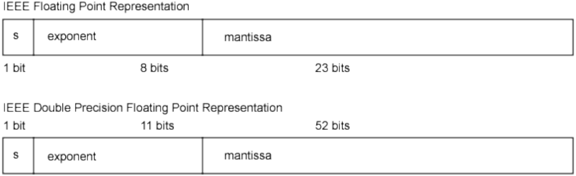
Challenge of the day

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

- What the hell?

Floating point representation

IEEE 754 standard



- 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

$$x = s \times m \times 2^{e-b} \tag{1}$$

A simple example

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

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

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

```
0.75 -> 0x3F400000 = 0b|0|01111110|10000000000000000000000
sign = 0b0 = 0 -> +
exponent = 0b01111110 = 126 -> 126 - 127 = -1
significand = 0b(1)10000000000000000000000 = 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

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

Floating point representation in Python

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

```
a = 0.75
num, den = a.as_integer_ratio()
print(num, den)
print('{:.30f}'.format(num / den))

print()

b = 0.1
num, den = b.as_integer_ratio()
print(num, den)
print('{:.30f}'.format(num / den))

[Output]
3 4
0.750000000000000000000000000000

3602879701896397 36028797018963968
0.100000000000000005551115123126
```

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

- 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