karthikraman / python-basics.ipynb Secret

Last active 3 months ago • Report abuse

☆ Star

Code Revisions 6 Stars 3 Forks 3

Tutorial: Python Basics

python-basics.ipynb

Quick Introduction to Python

A Sneak Peek at Programming in Python

Python is a versatile progamming language, and is particularly popular in scientific domains. In this notebook, we will try to understand some very basic elements of Python programming.

Contents

Installing Python

If you have not yet installed Python, check out the instructions here. The best way to use Python is by running iPython. This is a *notebook* that is written using iPython. The moment you fire up iPython, you will be able to interact with Python by typing various commands.

Getting Started

In [1]:

```python
print('Hello, world!!!')
```

```
Hello, world!!!
```

In [2]:

```python
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

`help` lets you get help on any in-built function, class etc. in Python. It is also easy to write help/documentation in Python, as we will see later. The special variable `__doc__` stores the documentation string in Python:

In [3]:

```python
print(print.__doc__)
```

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

Basic Python Types

- Primitive data types
- Integers (`int`)
- Floating point numbers (`float`)
- Strings (`str`)
- Booleans (`bool`)
- Built-in composite data types
- Lists (`list`)
- Tuples (`tuple`)
- Associative arrays (`dict`)
- User-defined data types: various objects can be created via `class` definitions

In [4]:

```
type(9.)
```

Out[4]:

```
float
```

In [5]:

```
type(3.1415)
```

Out[5]:

```
float
```

In [6]:

```
type('Hello!')
```

Out[6]:

```
str
```

In [7]:

```
type("Hello")
```

Out[7]:

```
str
```

In [8]:

```
type('''Hello
dlfdsf;alsdf
sdlafkdasfkjl''')
```

Out[8]:

```
str
```

In [9]:

```
type (False)
```

Out[9]:

```
bool
```

In [10]:

```
not False
```

Out[10]:

```
True
```

Built-in composite data types

In [11]:

```
type(__doc__)
```

Out[11]:

```
str
```

In [12]:

```
type([1,2,3,4,5])
```

Out[12]:

```
list
```

In [13]:

```
type((1,2))
```

Out[13]:

```
tuple
```

In [14]:

In [14]:

```
type((1))
```

Out[14]:

```
int
```

In [15]:

```
type((1,))
```

Out[15]:

```
tuple
```

In [16]:

```
len((1,))
```

Out[16]:

```
1
```

User-defined data types: objects can be created via class definitions

Simple Statements

- Assignment (=)
- `print`
- `return`
- `import`
- `pass`

In [17]:

```
courseNum = 3051
```

In [18]:

```
courseName = 'Data Structures and Algorithms for Biology'
```

In [19]:

```
print('BT',courseNum, ':', courseName)
```

```
BT 3051 : Data Structures and Algorithms for Biology
```

In [20]:

```
import math #Imports math libraries!
```

In [21]:

```
float(math.factorial(70))
```

Out[21]:

```
1.197857166996989e+100
```

In [22]:

```
math.factorial(70)
```

Out[22]:

```
11978571669969891796072783721689098736458938142546425857555362864628009582789
845319680000000000000000
```

In [23]:

```
pass #Do nothing!
```

In [24]:

```
math.pi
math.pi
```

Out[24]:
```
3.141592653589793
```

In [25]:
```
print('%s: %.40f' % ('pi',math.pi))
```

```
pi: 3.1415926535897931159979634685441851615906
```

In [26]:
```
math.e
```

Out[26]:
```
2.718281828459045
```

In [27]:
```
math.exp(1)
```

Out[27]:
```
2.718281828459045
```

In [28]:
```
dir(math)
```

Out[28]:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'trunc']
```

In [29]:

```
dir(__builtin__)
```

Out[29]:

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',
 'RecursionError',
 'ReferenceError',
 'ResourceWarning',
 'RuntimeError',
 'RuntimeWarning',
 'StopAsyncIteration',
 'StopIteration',
 'SyntaxError',
 'SyntaxWarning',
 'SystemError',
 'SystemExit',
 'TabError',
 'TimeoutError',
 'True',
 'TypeError',
 'UnboundLocalError',
 'UnicodeDecodeError',
 'UnicodeEncodeError',
 'UnicodeError',
 'UnicodeTranslateError',
 'UnicodeWarning',
 'UserWarning',
 'ValueError',
 'Warning',
 'WindowsError',
 'ZeroDivisionError',
 '__IPYTHON__',
 '__IPYTHON__active',
 '__build_class__',
 '__debug__',
 '__doc__',
 '__import__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'abs',
 'all',
 'any',
 'ascii',
 'bin',
 'bool',
 'bytearray',
 'bytes',
 'callable',
 'chr',
 'classmethod',
 'compile',
 'complex',
 'copyright',
 'credits',
 'delattr',
 'dict',
 'dir',
 'divmod',
 'dreload',
 'enumerate',
 'eval',
 'exec',
 'filter',
 'float',
 'format',
 'frozenset',
 'get_ipython',
 'getattr',
 'globals',
 'hasattr',
 'hash',
 'help',
 'hex',
 'id',
 'input',
 'int',
 'isinstance',
 'issubclass',
 'iter',
 'len',
 'license',
 'list',
 'locals',
 'map',
 'max',
 'memoryview',
 'min',
 'next',
 'object',
 'oct',
 'open',
 'ord',
 'pow',
 'print',
 'property',
 'range',
 'repr',
 'reversed',
 'round',
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Compound Statements

- `if`, `elif`, `else`
- Function definitions (`def`)
- Loops (`for`, `while`, `range`)

In [30]:

```
if math.factorial(5) == 120:
    print('Correct!')
```

```
Correct!
```

In [31]:

```
x=10
if x == 0:
    print(x)
else:
    print ('!')
```

```
!
```

In [32]:

```
def fac(x):
    if x==0:
        return 1
    else:
        return(x*fac(x-1))
```

In [33]:

```
fac(12)
```

Out[33]:

```
479001600
```

Strings in Python

Python has a battery of string functions, as can be seen below:

In [34]:

```
dir(str)
```

Out[34]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

`help(str)` will give you the entire documentation of the `str` class

String methods are very important for biology! Some important methods:

- `find()`

- `count()`
- `index()`
- `join()`
- `replace()`
- `split()`
- `strip()`
- `lower()`
- `upper()`
- `title()`

In [35]:

```
print(courseName[:5])
```

```
Data
```

In [36]:

```
courseName[5:]
```

Out[36]:

```
'Structures and Algorithms for Biology'
```

In [37]:

```
print(courseName)
```

```
Data Structures and Algorithms for Biology
```

In [38]:

```
x = input('>')
print('#',x,'#',sep='')
```

```
>     Karthik
#     Karthik    #
```

In [39]:

```
x.strip() #scrub whitespace around the string
```

Out[39]:

```
'Karthik'
```

Python has a rich repository of built-in functions!

In [40]:

```
dir(__builtin__)
```

Out[40]:

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',
 'RecursionError',
 'ReferenceError',
 'ResourceWarning',
 'RuntimeError',
 'RuntimeWarning',
 'StopAsyncIteration',
 'StopIteration',
 'SyntaxError',
 'SyntaxWarning',
 'SystemError',
 'SystemExit',
 'TabError',
 'TimeoutError',
 'True',
 'TypeError',
 'UnboundLocalError',
 'UnicodeDecodeError',
 'UnicodeEncodeError',
 'UnicodeError',
 'UnicodeTranslateError',
 'UnicodeWarning',
 'UserWarning',
 'ValueError',
 'Warning',
 'WindowsError',
 'ZeroDivisionError',
 '__IPYTHON__',
 '__IPYTHON__active',
 '__build_class__',
 '__debug__',
 '__doc__',
 '__import__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'abs',
 'all',
 'any',
 'ascii',
 'bin',
 'bool',
 'bytearray',
 'bytes',
 'callable',
 'chr',
 'classmethod',
 'compile',
 'complex',
 'copyright',
 'credits',
 'delattr',
 'dict',
 'dir',
 'divmod',
 'dreload',
 'enumerate',
 'eval',
 'exec',
 'filter',
 'float',
 'format',
 'frozenset',
 'get_ipython',
 'getattr',
 'globals',
 'hasattr',
 'hash',
 'help',
 'hex',
 'id',
 'input',
 'int',
 'isinstance',
 'issubclass',
 'iter',
 'len',
 'license',
 'list',
 'locals',
 'map',
 'max',
 'memoryview',
 'min',
 'next',
 'object',
 'oct',
 'open',
 'ord',
 'pow',
 'print',
 'property',
 'range',
 'repr',
 'reversed',
 'round',
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Again, `help(__builtin__)` will give you very detailed descriptions of each of the builtin types, classes, ...

In [41]:

```
print('Sadasd bbsad aslkdjsal aslad'.capitalize())
```

```
Sadasd bbsad aslkdjsal aslad
```

In [42]:

```
help('a'.capitalize)
```

```
Help on built-in function capitalize:

capitalize(...) method of builtins.str instance
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
```

File Handling

- `open()`
- `read()`
- `readlines()`
- `write()`
- `pickle()` !

In [43]:

```
f = open('names.txt')
```

In [44]:

```
x=f.read()
```

In [45]:

```
print(x.split('\n'))
```

```
['Alan Turing', 'Alan Perlis', 'Maurice Wilkes', 'Richard Hamming', 'Marvin Min
sky', 'James Wilkinson', 'John McCarthy', 'Edgser Dijkstra', 'Charles Bachman',
'Donald Knuth', '']
```

In [46]:

```
f = open('names.txt')
```

In [47]:

```
namesList = f.readlines()
```

In [48]:

```
print(namesList)
```

```
['Alan Turing\n', 'Alan Perlis\n', 'Maurice Wilkes\n', 'Richard Hamming\n', 'Ma
rvin Minsky\n', 'James Wilkinson\n', 'John McCarthy\n', 'Edgser Dijkstra\n', 'C
harles Bachman\n', 'Donald Knuth\n']
```

In [49]:

```
f = open('names.txt')
for name in f.readlines():
    print(name.strip())
f.close()
```

```
Alan Turing
Alan Perlis
Maurice Wilkes
Richard Hamming
Marvin Minsky
James Wilkinson
John McCarthy
Edgser Dijkstra
Charles Bachman
Donald Knuth
```

In [50]:

```
with open('names.txt') as f:
    for name in f.readlines():
        print(name.strip())
```

```
Alan Turing
Alan Perlis
Maurice Wilkes
Richard Hamming
Marvin Minsky
James Wilkinson
John McCarthy
Edgser Dijkstra
Charles Bachman
Donald Knuth
```

Python 3 vs Python 2

Since we're starting afresh, let's use Python 3

- However, codeskulptor supports only Python 2
- 3/2=1 (integer division!) in Python 2
- 3//2 is integer division in Python 3

- `3//2` is integer division in Python 3
- `3/2 = 1.5` in Python 3
- Does not work on codeskulptor though
- `print` is a function in Python 3
- `print (1)` instead of print 1

Exercise 1

- Check out the site Project Euler
- It has a number of mathematical problems to solve
- Solve a problem on Project Euler and post the solution on Piazza, via a codeskulptor URL
- Outcome: Basic Python use (loops, arithmetic etc.) and familiarity with codeskulptor

This brings us to the end of a whirlwind tour of some of the key features of Python. Here is a quote by Charles Severance on Python/programming: "*Programming is like learning an instrument, it takes practice. Don't expect to be able to play Bach on your first day. There is a steep learning curve when you start, but it gets easier quickly, just keep going. Feeling frustrated is fine, you are learning and it will make sense really soon. The thrill you get when it finally works is indescribable and you will be hooked. Python is an easy language to learn; it's easier than C++ or Objective C. I also think it's easier than Java. Javascript is probably about as easy and some courses choose to teach that. One of the advantages of Python is the number of resources there are to help you learn. Being able to get help when you get stuck is really important at the start. With Python, you can search online and there will usually be an answer. ... I don't think there is any sinister reason why Python is now taught in introductory computer science courses across the world, it's because it is easy for beginners to learn.*"

Python Basics

Variables

In [51]:

```
phi = 1.618034   # a floating point number
```

In [52]:

```
phi = 'golden ratio'   # a string
```

In [53]:

```
pingala = [1, 2, 3, 5, 8, 13, 21, 34]  # a list in Python
```

- Comments are indicated with `#`
- Triple quotes are used for documentation. (They MUST NOT BE USED for block comments!)

- Variables can change in value, *and type!* (unlike C/C++/Java)

Operators

Arithmetic Operators

```
In [54]: 2+4

Out[54]: 6

In [55]: 3*4

Out[55]: 12

In [56]: 2**4 #exponentiation

Out[56]: 16

In [57]: 2^4 #Not exponentiation, what is it?

Out[57]: 6

In [58]: 3/2

Out[58]: 1.5

In [59]: 3//2 #explicit integer division

Out[59]: 1
```

Comparison Operators

```
In [60]: 2 == 2

Out[60]: True

In [61]: 2 < 3

Out[61]: True

In [62]: 3 >= (1*3)

Out[62]: True
```

In [63]:

```
x = 1
```

In [64]:

```
0 < x < 2
```

Out[64]:

```
True
```

In [65]:

```
0 < x < 1
```

Out[65]:

```
False
```

In [66]:

```
0.1 + 0.2 == 0.3
```

Out[66]:

```
False
```

Oops, what happened!?

Aside: Floating Point Arithmetic

In [67]:

```
0.1+0.2
```

Out[67]:

```
0.30000000000000004
```

In [68]:

```
0.2+0.1+0.1+0.2
```

Out[68]:

```
0.6000000000000001
```

In [69]:

```
0.1+0.2+0.2+0.1
```

Out[69]:

```
0.6
```

Is addition really associative!?

Addition of floating point numbers is not associative! It is very wrong to check for the *exact value* of floating point numbers! Check if `abs(x1 - x2) < eps` or compare **significant figures**!

In [70]:

```
x1 = 0.1 + 0.2
x2 = 0.3
(x1 - x2)
```

Out[70]:

```
5.551115123125783e-17
```

In [71]:

```
0.1+0.1==0.2
```

Out[71]:

```
True
```

In [72]:

```
0.1+0.1+0.1==0.3
```

Out[72]:

```
False
```

In [73]:

```
x = 0
for i in range(10):
    x += 0.1
for i in range(10):
    x -= 0.1
print (x)
```

```
2.7755575615628914e-17
```

In [74]:

```
x = 0
for i in range(10):
    x += 0.125
for i in range(10):
    x -= 0.125
print (x)
```

```
0.0
```

In [75]:

```
1.0 + 2**(-52)
```

Out[75]:

```
1.0000000000000002
```

In [76]:

```
1.0 + 2**(-53)
```

Out[76]:

```
1.0
```

In [77]:

```
2**-53
```

Out[77]:

```
1.1102230246251565e-16
```

What is so special about 2^{-52}? It is the smallest number, which when added to a floating point 1.0, will change its value. It is also known as *machine epsilon*.

In [78]:

```
def machine_epsilon(prec=float):
    eps = 1.0
    while prec(1.) + prec(eps) != prec(1.):
        eps/=2
    return eps*2

print(machine_epsilon())
import numpy as np
print (machine_epsilon(np.float32))
print (machine_epsilon(np.float64))
```

```
2.220446049250313e-16
1.1920928955078125e-07
```

```
2.220446049250313e-16
```

Boolean Operations

In [79]:

```python
x = 1
```

In [80]:

```python
y = 5
```

In [81]:

```python
x is 1
```

Out[81]:

```
True
```

In [82]:

```python
y is not 5
```

Out[82]:

```
False
```

In [83]:

```python
x == 1 and y == 5
```

Out[83]:

```
True
```

In [84]:

```python
x != 1
```

Out[84]:

```
False
```

In [85]:

```python
not x == 1
```

Out[85]:

```
False
```

In [86]:

```python
x is not 1
```

Out[86]:

```
False
```

- Python uses `and` , `not` , `is` etc. instead of the symbols in many other languages
- Makes it more readable!
- Python does lazy verification of compound statements (short-circuit): e.g. `if x!=0 and n/x<0.5:`

Control Flow

In [87]:

```python
score = 75
if 85 < score <= 100:
    print('S')
```

```
elif 75 < score <= 85:
    print('A')
else:
    print('You must work harder!')
```

```
You must work harder!
```

Loops

while loop

In [88]:

```
def fib_gt_n(nmax):
    """Computes the first Pingala-Fibonacci number greater than a
    given number
    """
    fibn = 1
    fibn1 = 1
    n = 2
    while True:
        n = n+1
        fib = fibn+fibn1
        fibn1 = fibn
        fibn = fib
        if fib > nmax:
            break
    print ('F', n, ': ', fib, sep='')
```

In [89]:

```
fib_gt_n(1000)
```

```
F17: 1597
```

In [90]:

```
fib_gt_n(10000)
```

```
F21: 10946
```

for loop

`for` in Python is markedly different from other languages: in Python, `for` iterates over elements in an object (*iterable*); it's essentially a `for each` :

In [91]:

```
num = [1,2,3,4,5,6,7,8]
for i in num:
    print (i**i)
```

```
1
4
27
256
3125
46656
823543
16777216
```

A very important `iterable` is the `range` function:

In [92]:

```python
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [93]:

```python
list(range(5))
```

Out[93]:

```
[0, 1, 2, 3, 4]
```

In [94]:

```python
for i in range (4,-1,-1):
    print(i)
```

```
4
3
2
1
0
```

In [95]:

```python
for i in range(0,10,2):
    print(i)
```

```
0
2
4
6
8
```

In [96]:

```python
list(range(5))
```

Out[96]:

```
[0, 1, 2, 3, 4]
```

Functions

`def` ining a function

In [97]:

```python
def polyval(p, x):
    """Computes the value of a polynomial with specified coefficients p
    for a given value of x
    (list, float) -> float
    """
    value = 0
    i = 0
    for coeff in p:
        value += coeff * (x ** i)
        i = i + 1
    return value

polyval([1, 1, 1, 1],4)
```

Out[97]:

```
85
```

In [98]:

```
help(polyval)
```

```
Help on function polyval in module __main__:

polyval(p, x)
    Computes the value of a polynomial with specified coefficients p
    for a given value of x
    (list, float) -> float
```

In [99]:

```
print(polyval.__doc__)
```

```
Computes the value of a polynomial with specified coefficients p
    for a given value of x
    (list, float) -> float
```

In [100…

```
def gauss(n):
    """Sums the first n natural numbers
    """
    return sum(range(n+1))

gauss (100)
```

Out[100…

```
5050
```

In [101…

```
def print_grades(score):
    """ (int) --> NoneType
    >>> print_grades(70)
    You must work harder!
    >>> print_grades(90)
    S
    >>> print_grades(85)
    A
    """
    if 85<score<=100:
        print('S')
    elif 75<score<=85:
        print('A')
    else:
        print('You must work harder!')

a = print_grades(75)
print(a)
```

```
You must work harder!
None
```

Automatic Documentation and Testing in

Automatic Documentation and Testing in Python

Documentation and commenting

Read through this page too.

In [102…

```
help(print_grades)
```

```
Help on function print_grades in module __main__:

print_grades(score)
    (int) --> NoneType
    >>> print_grades(70)
    You must work harder!
    >>> print_grades(90)
    S
    >>> print_grades(85)
    A
```

Also, automatic testing!

In [103…

```
import doctest
```

In [104…

```
doctest.testmod(verbose = True)
```

```
Trying:
    print_grades(70)
Expecting:
    You must work harder!
ok
Trying:
    print_grades(90)
Expecting:
    S
ok
Trying:
    print_grades(85)
Expecting:
    A
ok
6 items had no tests:
    __main__
    __main__.fac
    __main__.fib_gt_n
    __main__.gauss
    __main__.machine_epsilon
    __main__.polyval
1 items passed all tests:
   3 tests in __main__.print_grades
3 tests in 7 items.
3 passed and 0 failed.
Test passed.
```

Out[104…

```
TestResults(failed=0, attempted=3)
```

Coding Style

In [105…

```python
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!
```

Did you notice?

- Python uses indentation to group statements/code blocks
- No braces `{}` like in C/C++
- Each code block, like an `if` or a `for` loop is indented by the same amount
- Python will throw a fit if the indentation is incorrect
- Always use [4] spaces for indentation
- Wrong indentation can also produce wrong code (if you are nesting)
- Style: Check out PEP8

Remember..

- Practice, practice, practice!
- Document your files/functions
- Comment your code
- Write test cases (make it a habit!)
- How to write good test cases?
- Learn python idioms (*pythonisms*)

Exercises

1a. Encode Newton-Raphson method for finding the zero of an arbitrary input function.

- Input: A Python function and an initial value `x0`
- Output: The value `x` at which the function goes to zero, or an error message if the zero could not be found

```
#!/usr/bin/python
def newton_raphson(f, x0):
    """Computes the zero of a function using the Newton-
Raphson
    method

    Args:
      func f
      float x0

    Returns:
      float x

    Examples:
      >>> newton_raphson (lambda(x) : x*x - 9, 2)
      3.0000
    """
    ...


    return x````

1b. (Do create an account on Rosalind) Solve the following
problems:
 * [Counting DNA nucleotides]
(http://rosalind.info/problems/dna/)
 * [Transcribing DNA into RNA]
(http://rosalind.info/problems/rna/)
```

Data Types in Python

Lists

A very detailed and excellent introduction to Lists, Strings and Tuples is here.

Lists store a *sequence* of data, which supports *indexing*

In [106…

```
primes = [2, 3, 5, 7, 11]
primes[0] #indices start at 0
```

Out[106…

```
2
```

In [107…

```
primes[-1]#negative indexing also works!
```

Out[107…

```
11
```

In [108…

```python
primes[:3]
```

Out[108…

```
[2, 3, 5]
```

In [109…

```python
primes[3:]
```

Out[109…

```
[7, 11]
```

In [110…

```python
primes[:-1]
```

Out[110…

```
[2, 3, 5, 7]
```

In [111…

```python
import sys
try:
    primes[5]
except:
    print(sys.exc_info())
```

```
(<class 'IndexError'>, IndexError('list index out of range',), <traceback objec
t at 0x000001B6744963C8>)
```

In [112…

```python
primes.append(13)
primes
```

Out[112…

```
[2, 3, 5, 7, 11, 13]
```

In [113…

```python
a = primes.pop()
a
```

Out[113…

```
13
```

In [114…

```python
len(primes)
```

Out[114…

```
5
```

In [115…

```python
max(primes)
```

Out[115…

```
11
```

In [116…

```python
3 in primes
```

Out[116…

```
True
```

In [117…

```python
for p in primes:
    print (p)
```

```
2
3
5
7
11
```

Lists can be easily concatenated, using the `+` operator:

In [118…

```
[1, 2, 3] + [4, 5, 6]
```

Out[118…

```
[1, 2, 3, 4, 5, 6]
```

In [119…

```
[0]*10
```

Out[119…

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Enumerating a list

In [120…

```
for i, val in enumerate(primes):
    print(i, val)
```

```
0 2
1 3
2 5
3 7
4 11
```

We don't have to do something boring and ugly like

```
i = 0
for val in primes:
    print (i, val)
    i = i + 1
```

List Comprehensions

This is a very powerful construct in python, to create new lists by manipulating existing ones.

In [121…

```
squares = [v*v for v in range(15)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

In [122…

```
squares_9 = [v*v for v in range(15) if (v*v)%9==0 ]
print(squares_9)
```

```
[0, 9, 36, 81, 144]
```

Nested comprehensions are also possible (very expressive)!

Nested comprehensions are also possible (very *expressive*)!

In [123...

```
set_x = list(range(5))
set_y = list(range(2))
set_x_cross_y = [[x,y] for x in set_x for y in set_y]
print(set_x_cross_y)
```

```
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1], [3, 0], [3, 1], [4, 0], [4,
1]]
```

In [124...

```
for i in range(1,31):
    for j in range(i,31):
        for k in range (j,31):
            if i*i+j*j==k*k:
                print(i,j,k)
```

```
3 4 5
5 12 13
6 8 10
7 24 25
8 15 17
9 12 15
10 24 26
12 16 20
15 20 25
18 24 30
20 21 29
```

In [125...

```
[(x,y,z) for x in range(1,31) for y in range (x,31) for z in range(y,31) if x
```

Out[125...

```
[(3, 4, 5),
 (5, 12, 13),
 (6, 8, 10),
 (7, 24, 25),
 (8, 15, 17),
 (9, 12, 15),
 (10, 24, 26),
 (12, 16, 20),
 (15, 20, 25),
 (18, 24, 30),
 (20, 21, 29)]
```

Tuples

A tuple is an ordered list of values, separated by commas:

In [126...

```
t = 12345, 54321, 'hello!'
t
```

Out[126...

```
(12345, 54321, 'hello!')
```

In [127...

```
t[0]
```

Out[127...

```
12345
```

Out[127... 12345

In [128...

```
# Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
u
```

Out[128...

```
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values `12345, 54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

In [129...

```
x, y, z = t
print(x,y,z,sep='\n')
```

```
12345
54321
hello!
```

Tuples are similar to lists and strings, but they are **immutable** --- i.e. they cannot be modified:

In [130...

```
x0 = (0, 0, 0)
```

In [131...

```
x0[0]
```

Out[131...

```
0
```

In [132...

```
import sys
try:
    x0[0] = 1
except:
    print(sys.exc_info())
```

```
(<class 'TypeError'>, TypeError("'tuple' object does not support item assignmen
t",), <traceback object at 0x000001B674494DC8>)
```

In [133...

```
x, y, z = x0
[x, y, z]
```

Out[133...

```
[0, 0, 0]
```

Tuples and String Formatting

Tuples pack a number of values for formatting strings, similar to `printf` statements in C/C++.

In [134...

```
print ('The %sth and %dth digits of pi are %d and %d%%.' % ('hundred', 1000,
The hundredth and 1000th digits of pi are 9 and 9%.
```

Strings

- Strings are similar to lists --- they can be indexed and sliced
- A horde of built-in commands are available --- recall `dir (str)`

In [135…

```
msg = 'Hello, world!'
```

In [136…

```
msg
```

Out[136…

```
'Hello, world!'
```

In [137…

```
msg[0:5]
```

Out[137…

```
'Hello'
```

In [138…

```
import sys
try:
    msg[0]='h'
except:
    print(sys.exc_info())
```

```
(<class 'TypeError'>, TypeError("'str' object does not support item assignmen
t",), <traceback object at 0x000001B674494608>)
```

In [139…

```
try:
    msg[-1] = '#'
except:
    print(sys.exc_info())
```

```
(<class 'TypeError'>, TypeError("'str' object does not support item assignmen
t",), <traceback object at 0x000001B674496A88>)
```

In [140…

```
for w in msg:
    print(w)
```

```
H
e
l
l
o
,
 
w
o
r
l
d
!
```

In [141…

```
msg[-1]
```

Out[141…

```
'!'
```

In [142…

```
msg[:5]
```

Out[142…

```
'Hello'
```

In [143…

```
msg[:5]+msg[5:]
```

Out[143…

```
'Hello, world!'
```

Parsing Strings

In [144…

```
data = '(1, 2, 3, 4, 5 , 6)'
```

In [145…

```
data = data.lstrip('(')
data
```

Out[145…

```
'1, 2, 3, 4, 5 , 6)'
```

In [146…

```
data = data.rstrip(')')
data
```

Out[146…

```
'1, 2, 3, 4, 5 , 6'
```

In [147…

```
nums = data.split(',')
nums
```

Out[147…

```
['1', ' 2', ' 3', ' 4', ' 5 ', ' 6']
```

In [148…

```
iData = [int(n) for n in nums]
iData
```

Out[148…

```
[1, 2, 3, 4, 5, 6]
```

In [149…

```
data = '(1, 2, 3, 4, 5 , 6)'
data
```

Out[149…

```
'(1, 2, 3, 4, 5 , 6)'
```

In [150…

```
#In a single line!
print([int(n) for n in data.strip('()').split(',')])
```

```
[1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

In [151…

```
len(msg)
```

Out[151…

```
13
```

In [152…

```
msg.upper()
```

Out[152…

```
'HELLO, WORLD!'
```

In [153…

```
'!!!'.join(['Hello', 'World!', 'Bye.'])
```

Out[153…

```
'Hello!!!World!!!!Bye.'
```

Dictionaries

- Very powerful data structure!
- Also known as *associative arrays*
- Maps from a set of keys to a set of values $K = \{ \text{ keys } \}$, $V = \{ \text{ values } \}$. $D : K \to V$:

$$k \xmapsto{D} v_k \in V$$

- In Python, you create D by a series of insertions of tuples $(k, v_k) \in K \times V$.
- It is *fast* to compute $D(k)$
- Dictionaries are particularly useful in biology!
- Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be ***any non-mutable type***; strings and numbers can always be keys
- Tuples *can be used* as keys if they contain only **strings, numbers, or tuples**
- Can't use `lists` as keys, since lists can be modified in place using `append()`
- Dictionary is an unordered set of `key:value` pairs --- keys must be unique

A pair of braces creates an empty dictionary:

In [154…

```
amino = {}
amino
```

Out[154…

```
{}
```

A comma-separated list of `{key:value}` pairs within the braces adds initial `key:value` pairs to the dictionary; dictionaries are also displayed in the same way on output.

In [155...

```
amino = {'A' : 'Ala', 'R': 'Arg', 'K': 'Lys', 'F':'Phe'}
```

In [156...

```
amino
```

Out[156...

```
{'A': 'Ala', 'F': 'Phe', 'K': 'Lys', 'R': 'Arg'}
```

Did you note, the keys are stored in *arbitrary order*?

Main operations on dictionaries

In [157...

```
'-'.join(amino[x] for x in 'FRAAAAFRAAAKA')
```

Out[157...

```
'Phe-Arg-Ala-Ala-Ala-Ala-Phe-Arg-Ala-Ala-Ala-Lys-Ala'
```

In [158...

```
amino['R']
```

Out[158...

```
'Arg'
```

In [159...

```
import sys
try:
    amino['S']
except:
    print(sys.exc_info())
```

```
(<class 'KeyError'>, KeyError('S',), <traceback object at 0x000001B6744AFA08>)
```

In [160...

```
amino['W'] = 'TRP'
```

In [161...

```
amino['W'] = amino['W'].title()
```

In [162...

```
amino
```

Out[162...

```
{'A': 'Ala', 'F': 'Phe', 'K': 'Lys', 'R': 'Arg', 'W': 'Trp'}
```

In [163...

```
del (amino['R'])
```

In [164...

```
amino
```

Out[164...

```
{'A': 'Ala', 'F': 'Phe', 'K': 'Lys', 'W': 'Trp'}
```

In [165...

```
amino.items()
```

Out[165…

```
dict_items([('K', 'Lys'), ('W', 'Trp'), ('F', 'Phe'), ('A', 'Ala')])
```

In [166…

```
amino.keys()
```

Out[166…

```
dict_keys(['K', 'W', 'F', 'A'])
```

In [167…

```
'K' in amino
```

Out[167…

```
True
```

In [168…

```
'G' in amino
```

Out[168…

```
False
```

In [169…

```
COMPLEMENT = {'A':'T', 'T':'A', 'C':'G', 'G':'C'}
dna = 'ATTAGCGCTTA'
cdna = [COMPLEMENT[base] for base in dna]
cdna
```

Out[169…

```
['T', 'A', 'A', 'T', 'C', 'G', 'C', 'G', 'A', 'A', 'T']
```

In [170…

```
''.join(reversed([COMPLEMENT[base] for base in dna]))
```

Out[170…

```
'TAAGCGCTAAT'
```

Exercise

1. Repeat the above exercise without using dictionaries
2. Can you find the three most frequent words in some Wikipedia article?

Modules

In [171…

```
import math
dir (math)
```

Out[171…

```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'trunc']
```

In [172…

```
math.pi
```

Out[172…

```
3.141592653589793
```

In [173…

```
from math import pi
pi
```

Out[173…

```
3.141592653589793
```

We can change the name of the module locally:

In [174…

```
import math as m
m.factorial(40)
```

```
Out[174…  815915283247897734345611269596115894272000000000
```

```
In [175…  m.pow(2,3)
```

```
Out[175…  8.0
```

```
In [176…  m.pow(pi,2)
```

```
Out[176…  9.869604401089358
```

It is possible to import everything from a module into the current namespace:

```
In [177…  from math import *
          cos(pi)
```

```
Out[177…  -1.0
```

However, One must always import only the needed functions! It is a crime to import everything, since we may end up with variable conflicts etc. However, it is useful, when you write your own modules...

__main__ in Python