Basic Python

Mads M. Pedersen (mmpe@dtu.dk)
David Robert Verelst (dave@dtu.dk)

Python 3 compatibility

Use this header in Python 2 to make python 3 compatible code:

```
from __future__ import division, print_function, absolute_import, unicode_literals
try: range = xrange; xrange = None
except NameError: pass
try: str = unicode; unicode=None
except NameError: pass
```

Print-function

```
print ("hello")
print (4+5)
print ("Hello: " + name)
print (4, 5)  # commas separate values by space
print ("hello: " + str(4+5))
```

Comments

```
#one line comment

"""
Multiline
comment
"""
```

Boolean

```
True, False
```

Operator	name	example	output
and	Logical conjunction	True and False	False
or	Logical disjunction	True or False	True
not	Negation	not True	False

Comparison

<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!= or <>	not equal
is	object identity
is not	negated object identity

Numeric types

Integer: 1, int(1.), int('1')
Long: 1L but don't care. Python does the job
Float: 1., float(1), float('1')

command	name	example	output
+	Addition	4+5	9
-	Subtraction	8-5	3
*	Multiplication	4*5	20
/	Division	19/3 5./10	6.333* .5
//	Floor division	19//3 5.//10	6 0
%	Remainder	19%3	5
**	Exponent	2**4	16

*Without "from __future__ import division" the result will be 6 in python 2

Statements

assert	Assert some condition	assert inp>0, "Input must be greater than 0)
pass	Does nothing but maintains indentation	If 1: pass

import	Import a module	import math
		math.sqrt(9)
		
		from math import sqrt
		sqrt(9)
		
		from math import * # use this with care
		sqrt(9)

exec	Execute a string	exec("s=5") s -> 5

Commonly used built in functions

abs()	Absolute value	abs(-4) -> 4
all()	All elements true	all([True,1,[False]) -> True
any()	Any element true	any([False, 0, [], None]) -> False
bin()	Binary representation	bin(8) -> 0b1000
bool()	Boolean representation	bool(3) -> True
chr()	Ascii character	chr(97) -> 'a'
complex()	Complex number	complex('1+2j') ->(1+2j)
dir()	List variables and functions	dir() -> ['__builtins__', '__doc__', ...] dir(8) -> ['__abs__', '__add__', ...]
eval()	Evaluate a string	eval('eval') -> <built-in function eval>

filter()	Filter a list	Filter(lambda x : x>=2, [1,2,3]) -> [2,3]
getattr()	Get an attribute of an object	getattr(8,'__abs__') -> abs function
globals()	Current global symbols	
hasattr()	Check if object has attribute	hasattr(8,'__abs__') -> True
hex()	Hexadecimal representation	hex(255) -> 0xff
input()	Python 3: Read line from input Python 2: read and eval line from input	>> s = input ("Your name: ") Your name: *Monty* >> s Monty
isinstance()	Check if object is instance of class	isinstance(8, int) -> True
locals()	Current local symbols	
map()	Apply a function to a list	map(lambda x : x*2, [1,2,3]) -> [2,4,6]
max()	Max value of a list	max([2,5,3]) -> 5
min()	Min value of list	min([2,5,2]) -> 2
oct()	Octal represenetation	oct(10) ->012
open()	Opens a file	open("txt.txt", 'r') -> file object
ord()	Index of ascii character	ord('a') -> 97
raw_input()	Python 2: Read line from input Python 3: Renamed to input	
reduce()	Reduce a list via (a,b)->c function	reduce(lambda a,b : a+b, [1,2,3]) -> 6
round()	Round a number	round(0.5) -> 1 round(-0.5) -> -1 round(1.25,1) -> 1.3
setattr()	Set an attribute of an object	
str()	String representation of object	str(8) -> "8"
sum()	Sum of list	sum([1,2,3]) -> 6

Sequences (String, list, tuple)

String: `"This is a string", 'this is a string', str(1)`

List: `[1,"a",(),[]]`

Tuple: `(1,"a",(),[])` Note: (1) is the number 'one' while (1,) is a tuple

Tuples are similar to lists but cannot be modified after creation

Command	Description	Example	Result
x in s	True if an item of s is equal to x, else False	1 in [1,2,3]	True
x not in s	False if an item of s is equal to x, else True	'a' in 'bcde'	True
s + t	the concatenation of s and t	(1,)+(2,)	(1,2)
s.append(x)	Append element x to list s	[1,2].append(3)	[1,2,3]
s.extend(t)	Extend list s with list t	[1].extend((2,3))	[1,2,3]
s.pop()			
s.remove(x)			
s * n	n copies of s concatenated	'a'*3	'aaa'
s[i]	ith item of s. First item is s[0]	'abc'[1]	b
S[-1]	ith item of s from the end. Last item is s[-1]	'abc'[-2]	b
s[i:j]	slice of s from i to j	[1,2,3,4][1:3]	[2,3]
s[i:j:k]	slice of s from i to j with step k	[1,2,3,4,5][1:-1:2] [1,2,3,4,5][::-1]	[2,4] [5,4,3,2,1]
len(s)	length of s	Len([1,2,3])	3
min(s)	smallest item of s	min("abc")	'a'
max(s)	largest item of s	Max((1,2,3))	3
s.index(i)	index of the first occurence of i in s	[1,2,3].index(2)	1
s.count(i)	total number of occurences of i in s	'this is it'.count('i')	3
reversed(s)	Reverse the sequence	reverse([1,2,3])	[3,2,1]
sorted(s)	Sort the sequence	Sorted([3,2,4])	[1,2,3]

String methods

Command	Description	Example	Result
str.upper()	Convert to upper case	'abc'.upper()	'ABC'
str.lower()	Convert to lower	'ABC'.upper()	'abc'
str.replace(x,y)	Replace	'abc'.replace('b','d')	'adc'
str.split(x)	Split to list of strings	'ab.cd.ef'.split('.')	['ab', 'cd', 'ef']
str.join(list)	Joins list of strings	','.join(['ab', 'cd', 'ef'])	'ab,cd,ef'

String formatting:

Include variable	'The value is: %s" % value
Include variables	'The values are: %s and %s" % (value1, value2)
Special characters	' \'quote\' '
New line	'\n'
Tab	'\t'

Formatting number variables

format_spec	[sign][#][0][width][.precision][type]	Example	Result
sign	"+": sign always shown	"%+d"%3 "%+d"%-3	'+3' '-3'
	"-": only negative sign shown (default)	"%-d"%3 "%-d"%-3	'3' '-3'
	" ": add space for positive numbers	"% d"%3 "% d"%-3	' 3' '-3'
#	Prefixes oct and hex values with '0o' or '0x	"%#x"%3	'0x3'
0	Zero padding instead of space padding	"%02d"%3	'03'
width	Number of pads before decimal separator	"%2d"%3	' 3'
precision	Number of space or 0 after decimal separator	"%.2f"%3	'3.00'
type	"d" decimal	"%d"%3	'3'
	"e" exponent notation	"%.2e"%3	'3.00e+00'
	"E" exponent notation	"%.2E"%3	'3.00E+00'
	"f", "F" fixed point	"%.2f"%3	'3.00'
	"g", "G" round/exponent notation if required	"%1.2g"%3.4 "%1.2g"%.456 "%1.2g"%234	'3.4' '0.46' '2.3e+02'
	"o" octal	"%#o"%9	'011'
	"x", "X" hex	"%#x"%9	'0x9'

Formatting string variables

format_spec	[align][width][type]	Example	Result
align	"+": right align(default) "-": left align	"%+3s"%2 "%-3s"%2	' 2' '2 '
width	String length	"%6s"%2	' 2'
type	"c" Unicode character	"%c"%2	'\x02'
	"s" String	"%s"%234	'234'

Set

An unordered collection of distinct hashable objects

```
s = set([1, 2, 3])
s = {1, 2, 3}
```

Command	Example	Description
Contains	Key in s	Return True if d contains key else False
Remove element	s.remove(e)	Remove e from s. If e not exists an error is thrown
Union	s1.union(s2) s1 | s2	
Intersection	s1.intersection(s2) s1 & s2	
Difference	s1.difference(s2) s1 – s2	

Dictionary

Mapping data structure where keys are an unordered collection of distinct hashable objects

```
d = dict(one=1, two=2, three=3)
d = {'one': 1, 'two': 2, 'three': 3}
d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
```

Command	Example	Description
Assign	d[key] = value	Assign value to key (replace if key exists)
Contains	Key in d	Return True if d contains key else False
Loop up	d[key]	Return value assigned to key. Throws error if key not exists
Get	d.get(key, default=None)	Return value assigned to key. Return default if key not exists
Delete	del d[key]	Delete key from d. If key not exists an error is thrown
Key list	d.keys()	Returns a list with the keys (in arbitrary order)
Value list	d.values()	Returns a list with the values (in arbitrary order)
Item list	d.items()	Returns a list with the (key, value) tuples (in arbitrary order)

When to use: Use dict instead of list if you need to look up values, as the running time of look up is proportional to the length of the list while expected constant for dictionaries.

Branching

```
if x==0:
    x=1
elif x==1:
    x=2
else:
    x=3
```

Unbound loop

```
while x<10:
    x+=1
```

The `pass` statement does nothing except maintaining the correct indent

Bound loop

```
for x in [0,1,2]:
    print x
```

```
for x in range(3):
    print x
```

Break and Continue

```
x = 0
while 1:
    x += 1
    if x % 2 == 0:
        continue
    if x > 10:
        break
    print x
# prints 1 3 5 7 9
```

Loop with branching

```
for i in range(10):
    if something_happens:
        ...
else:
    #something never happened
    ...
```

Iterators

range(x)	List from 0 to x	range(3)	[0,1,2]
range(x,y)	List form x to y	range(1,3)	[1,2]
range(x,y,z)	List from x to y step z	range(2,10,3)	[2,5,8]
zip(l1, l2,…)	Generates tuples with the i^{th} element from each list	zip((0,1),('a','b'))	[(0,'a'),(1,'b')]
enumerate(lst)	"Adds index to the list"	enumerate(['a','b'])	[(0,'a'),(1,'b')]
enumerate(lst, start)	"Adds index starting at <start> to the list"	enumerate(['a','b'],4)	[(4,'a'),(5,'b')]

In Python 2.x use xrange instead of range (xrange generates list on the fly(lazy evaluation) while range generates and saves the whole list in memory)

Three ways to do the same

`for x,y in [(0,'a'),(1,'b')]:`
`    print x,y`
	`for x,y in zip((0,1),('a','b')):`
	`    print x,y`
		`for x,y in enumerate(('a','b')):`
		`    print x,y`

List comprehension

Fast inline method to build lists

[x for x in range(3)]	[0,1,2]
[x+5 for x in range(3)]	[5,6,7]
[x for x in range(5) if x%2==0]	[0,2,4]
[(x,y) for x in range(2) for y in range(3)]	[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]

Generators

Lazy evaluation version of list comprehension, i.e. next element is created when needed

(x for x in range(3))	<generator object <genexpr> at 0x005C3DC8>
(x+5 for x in range(3))	<generator object <genexpr> at 0x005C3DC8>
(x for x in range(5) if x%2==0)	<generator object <genexpr> at 0x005C3DC8>
((x,y) for x in range(2) for y in range(3))	<generator object <genexpr> at 0x005C3DC8>

Usage:

`for x in <generator>:`
`    …`
	`list(<generator>)`	`It = iterator(<generator>)`
		`next(it)`
		`next(it)`

Custom generators:

```
def my_generator(n):
    i = 0
    while i<n:
        yield(i) #wait until 'next()'
        i+=1
```

Functions

```
def x_n(x, n=2):
    return x*n
```

x_n(2)	4
x_n(2,3)	6
x_n(x=2,n=3)#ok x_n(x,n=3) #ok x_n(x=2,3) #SyntaxError: non-keyword arg after keyword arg	
s = (2,3) x_n(*s) # * unpacks sequence s	6
d = {'x':2,'n':3} x_n(**d) # ** unpacks dict d	6
x_n('hi')	'hihi'
x_n	<function __main__.x2>

Global variables

```
v1 = 1
v2 = 2
def f():
    global v1
    v1 = 0 #assigment to global v1
    v2 = 0 #assignment to local v2
f()
print (v1, v2)
```

```
0, 2
```

Lamba functions

Inline function:

```
lambda <input> : <one line function body>
```

x_n = lambda x, n=2 : x*n	x_n(2)	4
	x_n(2,3)	6
	x_n	<function __main__.<lambda>>
	x_n(x_n(2))	8

x_n(x) executes the function

x_n is a variable holding a reference to the lambda function

x_n can be used as argument for a function, see below

*args, **kwargs

In function declaration:

'*args': unnamed arguments collected in args tuple
'**kwargs': named arguments collected in kwargs dict

```
def f(arg1, *args, **kwargs):
    print('arg1:', arg1)
    print('args:', args)
    print('kwargs:', kwargs)

f(1, 2, 3, a=4, b=5)
```

```
1
(2, 3)
{'a': 4, 'b': 5}
```

In function call:

'*args': unpack list or tuple
'**kwargs': unpack dict

```
def f(a1, a2, a3, a4):
    print (a1, a2, a3, a4)

a_list = [1, 2]
a_dict = {'a3':3, 'a4':4}
f(*a_list, **a_dict)
```

```
1 2 3 4
```

Note: All names can be used but args and kwargs are commonly used

Function as arguments

```
x_n = lambda x, n=2 : x*n

def f(func, *args,**kwargs):
    return func(*args,**kwargs)
```

f(x_n, (2,))	4
f(x_n,(2,3))	6
f(x_n, (x_n(2),))	8

print [func(2,3) for func in (x_n,min,max,sum)]	[6, 2, 3, 5]

Object-oriented

Aim at:

- Encapsulation: Hide the details
- Decoupling: Divide your program into coherent reusable and decoupled classes
- No code duplication: Reuse your classes or extend them via inheritance

Classes and objects

Class: Object template holding member variables and functions

Object: Class instance holding instance variables and reference to its class

```
class MyClass(object):
    pass

myClass = MyClass()
print (MyClass)  #class
print (myClass)  #object
print (myClass.__class__)
```

```
<class '__main__.MyClass'>
<__main__.MyClass object at 0x02290110>
<class '__main__.MyClass'>
```

Variable scope

```
v1 = 1  # global variable

class MyClass(object):
    v2 = 2  # member variable (copied to instance at instantiation)
    def __init__(self):
        self.v3 = 3  # instance variable
        self._v4 = 4  # instance variable

    def vars(self):
        return (v1, self.v2, self.v3, self._v4)

myClass = MyClass()
print (myClass.vars())
print (myClass._v4)  # Intended private: Do not access outside class
```

Function scope

```
class MyClass(object):
    def vars(self):
        return "vars"

    def _vars(self):
        # Intended private: Do not use outside class
        return "_vars"

    def __vars(self):
        # Intended private: Do not use outside class
        return  "__vars"

myClass = MyClass()
print (myClass.vars()) # prints: "vars"
print (myClass._vars()) # prints: "_vars"
print (myClass._MyClass__vars()) # prints: "__vars"
print (myClass.__vars())#AttributeError: 'MyClass' object has no attribute '__vars'
```

Inheritance

```
class A(object):
    def v(self):
        return "A"

class B(object):
    def v(self):
        return "B"

class C(A, B):
    def p(self):
        print (self.v(), B.v(self))


c = C()
c.p()  # prints "A B"
print (isinstance(c, A)) # prints True
print (isinstance(c, B)) # prints True
print (isinstance(c, C)) # prints True
```

Special methods

Method	Invoked by	Comment
__init__	MyClass()	Constructor
__str__	str(myObj)	Informal string representation of the object
__repr__	repr(myObj)	Official string representation
__call__	myObj ()	Makes the object callable
__getitem__	myObj [3:4:2]	
__setitem__	myObj [3:4:2] = v	
__len__	len(myObj)	
__del__	del myObj	Called when instance is about to be destroyed
__new__	MyClass()	Creates a new object of MyClass, i.e. copies member variables etc., and invokes the constructor of the new instance
__lt__	myObj < v	Comparison methods
__le__	myObj <= v	
__eq__	myObj == v	
__ne__	myObj != v	
__gt__	myObj > v	
__ge__	myObj >= v	
__add__	myObj + v	Operators
__sub__	myObj - v	
__mul__	myObj * v	
__floordiv__	myObj // v	
__mod__	myObj % v	
__pow__	myObj ** v	
__lshift__	myObj << v	Bitwise operators
__rshift__	myObj >> v	
__and__	myObj & v	
__xor__	myObj ^ v	
__or__	myObj | v	

Properties

Use properties instead of getters and setters.
In this way you can switch between variable and method representation while keeping the call syntax without ()

No need for method	Method required	Alternative syntax

class Time(object):
    sec = 120
 	
print (Time().sec)
	class Time(object):
	    min = 2
	
	    @property
	    def sec(self):
	        return self.min * 60
	
	    @seconds.setter
	    def seconds(self, sec):
	        self.min = sec / 60
	
	print (Time().sec)
		class Time(object):
		    min = 2
		
		    def gets(self):
		        return self.min * 60
		
		    def sets(self, sec):
		        self.min = sec / 60
		
		    sec = property(gets, sets)
		
		print (Time().sec)

Error handling

Dealing with I/O (file handling, user input etc.) is risky business.
Handle exceptions to avoid program termination, but handle with care, i.e. catch the lowest level exception to avoid masking errors that should be fixed instead.

```
try:
    x = int(raw_input("Please enter a number: "))
except ValueError:
    print ("Oops!  That was no valid number.")
except:
    print ("Unexpected error")
    raise # raise to higher level
```