Python 2.7 Quick Reference Sheet

ver 2.01 – 110105 (sjd)

Interactive Help in Python Shell

help()	Invoke interactive help
help(*m*)	Display help for module *m*
help(*f*)	Display help for function *f*
dir(*m*)	Display names in module *m*

Small Operator Precedence Table

func_name(*args, …*)	Function call
x[*index* : *index*]	Slicing
x[*index*]	Indexing
x.attribute	Attribute reference
**	Exponentiation
*, /, %	Multiply, divide, mod
+, -	Add, subtract
>, <, <=, >=, !=, ==	Comparison
in, not in	Membership tests
not, and, or	Boolean operators NOT, AND, OR

Module Import

import *module_name*
from *module_name* import *name* , …
from *module_name* import *

Common Data Types

Type	Description	Literal Ex
int	32-bit Integer	3, -4
long	Integer > 32 bits	101L
float	Floating point number	3.0, -6.55
complex	Complex number	1.2J
bool	Boolean	True, False
str	Character sequence	"Python"
tuple	Immutable sequence	(2, 4, 7)
list	Mutable sequence	[2, x, 3.1]
dict	Mapping	{ x:2, y:5 }

Common Syntax Structures

Assignment Statement

var = exp

Console Input/Output

var = input([*prompt*])
var = raw_input([*prompt*])
print *exp*[,] …

Selection

if (*boolean_exp*):
  stmt …
[elif (*boolean_exp*):
  stmt …] …
[else:
  stmt …]

Repetition

while (*boolean_exp*):
  stmt …

Traversal

for *var* in *traversable_object*:
  stmt …

Function Definition

def *function_name*(*parmameters*):
  stmt …

Function Call

function_name(*arguments*)

Class Definition

class *Class_name* [(*super_class*)]:
  [*class variables*]
  def *method_name*(self, *parameters*):
    stmt

Object Instantiation

obj_ref = *Class_name*(*arguments*)

Method Invocation

obj_ref.method_name(*arguments*)

Exception Handling

try:
  stmt …
except [*exception_type*] [, *var*]:
  stmt …

Common Built-in Functions

Function	Returns
abs(*x*)	Absolute value of *x*
dict()	Empty dictionary, eg: d = dict()
float(*x*)	int or string *x* as float
id(*obj*)	memory addr of *obj*
int (*x*)	float or string *x* as int
len(*s*)	Number of items in sequence *s*
list()	Empty list, eg: m = list()
max(*s*)	Maximum value of items in *s*
min(*s*)	Minimum value of items in *s*
open(*f*)	Open filename *f* for input
ord(*c*)	ASCII code of *c*
pow(*x,y*)	x ** y
range(*x*)	A list of x ints 0 to *x* - 1
round(*x,n*)	float *x* rounded to *n* places
str(*obj*)	str representation of *obj*
sum(*s*)	Sum of numeric sequence *s*
tuple(*items*)	tuple of *items*
type(*obj*)	Data type of *obj*

Common Math Module Functions

Function	Returns (all float)
ceil(*x*)	Smallest whole nbr >= *x*
cos(*x*)	Cosine of *x* radians
degrees(*x*)	*x* radians in degrees
radians(*x*)	*x* degrees in radians
exp(*x*)	e ** *x*
floor(*x*)	Largest whole nbr <= *x*
hypot(*x, y*)	sqrt(*x* * *x* + *y* * *y*)
log(*x* [, *base*])	Log of *x* to *base* or natural log if *base* not given
pow(*x, y*)	x ** y
sin(*x*)	Sine of *x* radians
sqrt(*x*)	Positive square root of *x*
tan(*x*)	Tangent of *x* radians
pi	Math constant pi to 15 sig figs
e	Math constant e to 15 sig figs

Common String Methods

***S*.method()**	**Returns (str unless noted)**
capitalize	*S* with first char uppercase
center(*w*)	*S* centered in str *w* chars wide
count(*sub*)	int nbr of non-overlapping occurrences of *sub* in *S*
find(*sub*)	int index of first occurrence of *sub* in *S* or -1 if not found
isdigit()	bool True if *S* is all digit chars, False otherwise
islower() isupper()	bool True if *S* is all lower/upper case chars, False otherwise
join(*seq*)	All items in *seq* concatenated into a str, delimited by *S*
lower() upper()	Lower/upper case copy of *S*
lstrip() rstrip()	Copy of *S* with leading/ trailing whitespace removed, or both
split([*sep*])	List of tokens in *S*, delimited by *sep*; if *sep* not given, delimiter is any whitespace

Formatting Numbers as Strings

Syntax: "*format_spec*" % *numeric_exp*
***format_spec* syntax:** % *width.precision type*

- *width* (optional): align in number of colums specified; negative to left-align, precede with 0 to zero-fill
- *precision* (optional): show specified digits of precision for floats; 6 is default
- *type* (required): d (decimal int), f (float), s (string), e (float – exponential notation)
- Examples for x = 123, y = 456.789
  "%6d" % x -> . . . 123
  "%06d" % x -> 000123
  "%8.2f % y -> . . 456.79
  "8.2e" % y -> 4.57e+02
  "-8s" % "Hello" -> Hello . . .

Common List Methods

***L*.method()**	**Result/Returns**
append(*obj*)	Append *obj* to end of *L*
count(*obj*)	Returns int nbr of occurrences of *obj* in *L*
index(*obj*)	Returns index of first occurrence of *obj* in *L*; raises ValueError if *obj* not in *L*
pop([*index*])	Returns item at specified *index* or item at end of L if *index* not given; raises IndexError if *L* is empty or *index* is out of range
remove(*obj*)	Removes first occurrence of *obj* from *L*; raises ValueError if *obj* is not in *L*
reverse()	Reverses *L* in place
sort()	Sorts *L* in place

Common Tuple Methods

***T*.method()**	**Returns**
count(*obj*)	Returns nbr of occurrences of *obj* in *T*
index(*obj*)	Returns index of first occurrence of *obj* in *T*; raises ValueError if *obj* is not in *T*

Common Dictionary Methods

***D*.method()**	**Result/Returns**
clear()	Remove all items from *D*
get(*k* [,*val*])	Return *D*[*k*] if *k* in *D*, else *val*
has_key(*k*)	Return True if *k* in *D*, else False
items()	Return list of key-value pairs in *D*; each list item is 2-item tuple
keys()	Return list of *D*'s keys
pop(*k*, [*val*])	Remove key *k*, return mapped value or *val* if *k* not in *D*
values()	Return list of *D*'s values

Common File Methods

***F*.method()**	**Result/Returns**
read([*n*])	Return str of next *n* chars from *F*, or up to EOF if *n* not given
readline([*n*])	Return str up to next newline, or at most *n* chars if specified
readlines()	Return list of all lines in *F*, where each item is a line
write(*s*)	Write str *s* to *F*
writelines(*L*)	Write all str in seq *L* to *F*
close()	Closes the file

Other Syntax

Hold window for user keystroke to close:

```
raw_input("Press  <Enter> to quit.")
```

Prevent execution on import:

```
if __name__ == "__main__":
    main()
```

Displayable ASCII Characters

32	SP	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	105	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

'\0' = 0, '\t' = 9, '\n' = 10

Mémento v2.0.4

Python 3 Cheat Sheet

Latest version on :
https://perso.limsi.fr/pointal/python:memento

Base Types

integer, float, boolean, string, bytes

```
int 783  0  -192  0b010  0o642  0xF3
                  null  binary  octal  hexa
float 9.23  0.0  -1.7e-6
                       ×10^-6
bool True  False
str "One\nTwo"
    escaped new line
    'I\'m'
    escaped '
    Multiline string:
    """X\tY\tZ
    1\t2\t3"""
    escaped tab
bytes b"toto\xfe\775"
      hexadecimal  octal
```

immutables

Container Types

- **ordered sequences**, fast index access, repeatable values

```
list [1,5,9]   ["x",11,8.9]   ["mot"]   []
tuple (1,5,9)  11,"y",7.4     ("mot",)  ()
```

Non modifiable values (immutables) — *expression with just comas →* `tuple`

```
str bytes   (ordered sequences of chars / bytes)   ""  b""
```

- **key containers**, no *a priori* order, fast key acces, each key is unique

```
dictionary  dict {"key":"value"}   dict(a=3,b=4,k="v")   {}
(key/value associations)  {1:"one",3:"three",2:"two",3.14:"π"}
collection  set {"key1","key2"}   {1,9,3,0}   set()
```

keys=hashable values (base types, immutables…) `frozenset` *immutable set* *empty*

Identifiers

for variables, functions, modules, classes… names

`a…zA…Z_` followed by `a…zA…Z_0…9`

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

☺ `a toto x7 y_max BigOne`
☹ ~~`8y and for`~~

Variables assignment

`=`

1) evaluation of right side expression value
2) assignment in order with left side names

assignment ⇔ **binding** of a *name* with a *value*

```
x=1.2+8+sin(y)
a=b=c=0        assignment to same value
y,z,r=9.2,-7.6,0   multiple assignments
a,b=b,a        values swap
a,*b=seq       unpacking of sequence in
*a,b=seq       item and list
x+=3           increment ⇔ x=x+3
x-=2           decrement ⇔ x=x-2
x=None         « undefined » constant value
del x          remove name x
```

and `*=` `/=` `%=` …

Conversions

`type(`*expression*`)`

```
int("15") → 15
int("3f",16) → 63        can specify integer number base in 2nd parameter
int(15.56) → 15          truncate decimal part
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6      rounding to 1 decimal (0 decimal → integer number)
bool(x)  False for null x, empty container x , None or False x ; True for other x
str(x)→"…"   representation string of x for display (cf. formating on the back)
chr(64)→'@'  ord('@')→64   code ↔ char
repr(x)→"…"  literal representation string of x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
separator str and sequence of str → assembled str
   ':'.join(['toto','12','pswd']) → 'toto:12:pswd'
str splitted on whitespaces → list of str
   "words with  spaces".split() → ['words','with','spaces']
str splitted on separator str → list of str
   "1,4,8,2".split(",") → ['1','4','8','2']
sequence of one type → list of another type (via comprehension list)
   [int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Sequence Containers Indexing

for lists, tuples, strings, bytes…

```
negative index   -5  -4  -3  -2  -1
positive index    0   1   2   3   4
lst=[10, 20, 30, 40, 50]
positive slice    0   1   2   3   4   5
negative slice   -5  -4  -3  -2  -1
```

Items count
`len(lst)→5`

index from 0
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`

```
lst[0]→10   ⇒ first one      lst[1]→20
lst[-1]→50  ⇒ last one       lst[-2]→40
```

On mutable sequences (`list`), remove with `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]   lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]     lst[::-2]→[50,30,10]         lst[-3:-1]→[30,40]  lst[3:]→[40,50]
lst[::2]→[10,30,50]      lst[:]→[10,20,30,40,50]  shallow copy of sequence
```

Missing slice indication → from start / up to end.
On mutable sequences (`list`), remove with `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparators: `<` `>` `<=` `>=` `==` `!=`
(boolean results) ≤ ≥ = ≠

`a and b` logical and — *both simulta-neously*

`a or b` logical or — *one or other or both*

pitfall : `and` *and* `or` *return* *value* *of* `a` *or of* `b` *(under shortcut evaluation).*
⇒ *ensure that* `a` *and* `b` *are booleans.*

`not a` logical not

`True` `False` — True and False constants

Statements Blocks

```
parent statement:
    statement block 1…
    ⋮
parent statement:
    statement block2…
    ⋮
next statement after block 1
```

indentation !

configure editor to insert 4 spaces in place of an indentation tab.

Modules/Names Imports

module `truc` ⇔ *file* `truc.py`

```
from monmod import nom1,nom2 as fct
```
→*direct acces to names, renaming with* `as`

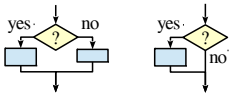
```
import monmod  →acces via monmod.nom1 …
```

modules and packages searched in *python path* (cf `sys.path`)

Conditional Statement

statement block executed only if a condition is true

`if` *logical condition*`:`
→ *statements block*



Can go with several *elif*, *elif*... and only one final *else*. Only the block of first true condition is executed.

```
if age<=18:
    state="Kid"
elif age>65:
    state="Retired"
else:
    state="Active"
```

with a var `x`*:*
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Maths

floating numbers… approximated values *angles in radians*

Operators: `+` `-` `*` `/` `//` `%` `**`
Priority `(…)` × ÷ ↑ ↑ a^b
integer ÷ ÷ remainder

`@` → matrix × *python3.5+* `numpy`

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

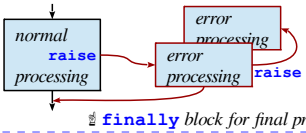
usual priorities

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0   √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

Exceptions on Errors

Signaling an error:
`raise` *Exception(…)*



Errors processing:

```
try:
    → normal procesising block
except Exception as e:
    → error processing block
```

`finally` *block for final processing in all cases.*

Conditional Loop Statement

*statements block executed **as long as** condition is true*

```
while logical condition:
    statements block
```


```
s = 0          # initializations before the loop
i = 1          # condition with a least one variable value (here i)
while i <= 100:
    s = s + i**2
    i = i + 1      # make condition variable change !
print("sum:",s)
```

Algo:

$$s = \sum_{i=1}^{i=100} i^2$$

beware of infinite loops!

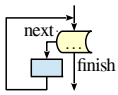
Loop Control

`break` *immediate exit*
`continue` *next iteration*
`else` *block for **normal** loop exit.*

Iterative Loop Statement

*statements block executed **for each** item of a container or iterator*

```
for var in sequence:
    statements block
```



Go over sequence's **values**

```
s = "Some text"    # initializations before the loop
cnt = 0
for c in s:        # loop variable, assignment managed by for statement
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Algo: count number of `e` *in the string.*

loop on dict/set ⇔ loop on keys sequences
use *slices* to loop on a subset of a sequence

Go over sequence's **index**

- modify item at index
- access items around index (before / after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Algo: limit values greater than 15, memorizing of lost values.

good habit : don't modify loop variable

Go simultaneously on sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

Display

```
print("v=",3,"cm :",x,",",y+4)
```

items to display : literal values, variables, expressions

`print` options:

- `sep=" "` items separator, default space
- `end="\n"` end of print, default new line
- `file=sys.stdout` print to file, default standard output

Input

```
s = input("Instructions:")
```

`input` always returns a **string**, convert it to required type (cf. boxed *Conversions* on the other side).

Generic Operations on Containers

*Note: For dictionaries and sets, these operations use **keys**.*

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` sorted *copy*
`val in c` → boolean, membership operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index, value)
`zip(c1,c2…)` → *iterator* on tuples containing c_i items at same index
`all(c)` → `True` if **all** `c` items evaluated to true, else `False`
`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

*Specific to **ordered sequences containers** (lists, tuples, strings, bytes…)*

`reversed(c)` → *inversed iterator* `c*5`→ duplicate `c+c2`→ concatenate
`c.index(val)` → *position* `c.count(val)` → *events count*

```
import copy
copy.copy(c)      → shallow copy of container
copy.deepcopy(c)  → deep copy of container
```

Operations on Lists

modify original list

`lst.append(val)`	add item at end
`lst.extend(seq)`	add sequence of items at end
`lst.insert(idx,val)`	insert item at index
`lst.remove(val)`	remove first item with value *val*
`lst.pop([idx])` →*value*	remove & return item at index *idx* (default last)
`lst.sort()` `lst.reverse()`	sort / reverse liste *in place*

Operations on Dictionaries

`d[key]=value` `d.clear()`
`d[key]` → *value* `del d[key]`
`d.update(d2)` update/add associations
`d.keys()` `d.values()` `d.items()` →*iterable views on keys/values/associations*
`d.pop(key[,default])` → *value*
`d.popitem()` → *(key,value)*
`d.get(key[,default])` → *value*
`d.setdefault(key[,default])` →*value*

Operations on Sets

Operators:
`|` → union (vertical bar char)
`&` → intersection
`-` `^` → difference/symetric diff.
`<` `<=` `>` `>=` → inclusion relations
Operators also exist as methods.

`s.update(s2)` `s.copy()`
`s.add(key)` `s.remove(key)`
`s.discard(key)` `s.clear()`
`s.pop()`

Files

storing data on disk, and reading it back

```
f = open("file.txt","w",encoding="utf8")
```

- file **variable** for operations
- **name** of file on disk (+path…)
- opening **mode**
  - `'r'` read
  - `'w'` write
  - `'a'` append
  - …`'+'` `'x'` `'b'` `'t'`
- **encoding** of chars for *text files*: `utf8` `ascii` `latin1` …

cf. modules `os`, `os.path` and `pathlib`

writing

```
f.write("coucou")
f.writelines(list of lines)
```

reading

read empty string if end of file

```
f.read([n])        → next chars
    if n not specified, read up to end !
f.readlines([n])   → list of next lines
f.readline()       → next line
```

*text mode `t` by default (read/write `str`), possible binary mode `b` (read/write `bytes`). **Convert from/to required type** !*

`f.close()` dont forget to **close the file** after use !

`f.flush()` write cache
`f.truncate([taille])` resize

reading/writing progress sequentially in the file, modifiable with:

`f.tell()` →*position*
`f.seek(position[,origin])`

Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

Integers Sequences

`range([start,] end [,step])`

start default 0, *fin* not included in sequence, *pas* signed default 1

`range(5)` → 0 1 2 3 4
`range(3,8)` → 3 4 5 6 7
`range(2,12,3)` → 2 5 8 11
`range(20,5,-5)` → 20 15 10
`range(len(seq))` → *sequence of index of values in seq*

range provides an immutable sequence of int constructed as needed

Function Definition

function name (identifier)
named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`fct`

result value of the call, if no computed result to return: `return None`

parameters and all variables of this block exist only *in* the block and *during* the function call (think of a "black box")

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`

**args variable positional arguments (→`tuple`), default values,*
***kwargs variable named arguments (→`dict`)*

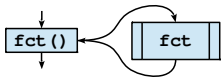
Function Call

```
r = fct(3,i+2,2*i)
```

storage/use of returned value
one argument per parameter

this is the use of function name *with parenthesis* which does the call

*Advanced: *sequence **dict*



Operations on Strings

`s.startswith(prefix[,start[,end]])` `s.endswith(suffix[,start[,end]])` `s.strip([chars])`
`s.count(sub[,start[,end]])` `s.partition(sep)` → *(before,sep,after)*
`s.index(sub[,start[,end]])` `s.find(sub[,start[,end]])`
`s.is…()` *tests on chars categories (ex. `s.isalpha()`)*
`s.upper()` `s.lower()` `s.title()` `s.swapcase()`
`s.casefold()` `s.capitalize()` `s.center([width,fill])`
`s.ljust([width,fill])` `s.rjust([width,fill])` `s.zfill([width])`
`s.encode(encoding)` `s.split([sep])` `s.join(seq)`

Formating

formating directives — values to format

```
"modele{} {} {}".format(x,y,r)  → str
```

`"{`*selection* `:` *formating* `!` *conversion*`}"`

- **Selection** :

```
2
nom
0.nom
4[key]
0[2]
```

Examples:

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="I'm")
→'"I\'m"'
```

- **Formating** :

fill char *alignment* *sign* *mini width* . *precision~maxwidth* *type*

`<` `>` `^` `=` `+` `-` *space* `0` at start for filling with 0

integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default),
string: `s` … `%` percent

- **Conversion** : `s` (readable texte) or `r` (literal representation)

Syntax

Python code has a simple syntax and is generally easy to read and reason about, this session is intended to make sure you know how to read any code you come across.

But first, the bad news

Three things people (I) hate(d) about Python syntax:

1. No pre-declaration (can make scope-related bugs hard to find).
2. Many things are statements (don't return a value) in Python that aren't in other languages. (E.g. `print`, `+=`)
3. Significant whitespace. Statements in the same block must be indented equally.

Feel free to hate these too, but be warned it's well-trod ground.

Example 1

```
continue_meditating = True
while continue_meditating:
    if True:
        continue
    elif False:
        raise SystemExit("Gah! Paradox!")
    else:
        continue_meditating = False

share_enlightenment()
```

You just saw:

`while`, `if`, `elif`, and `else` do not require parenthesis around the condition they evaluate or a curly brace, but *do* require the colon at the end of the line.
Indentation denotes code blocks.
marks comments.
`raise` raises exceptions (`try` and `except` are used for catching them).
Objects are instantiated without a `new` keyword.
Functions are called by following their name with parens, containing arguments if necessary.

Data Literals

```
number_list = [1, 2, 5] # "Three sir!"
```

```
number_list.append(3)

# Dictionaries map from keys to values
music = {"Aidan": "Beats", "Justin": "Grunge"}
music['Stephen'] = 'Sarcasm'

birth_year = ('Stephen', 1984) # Tuples are immutable
birth_year[1] = 1341 # Raises an exception
```

http://docs.python.org/library/stdtypes.html

Functions

```
def shout(message="Hi"):
    print "%s!" % message

shout() # Prints "Hi!"
shout("I love python")
shout(message="And keyword arguments")
```

Functions are defined with the `def` keyword, and arguments can specify default values. Additionally, arguments that specify defaults can be named at the call site. This is referred to as a *keyword* argument.

Classes

```
class LoudTalker(object):
    def say(self, message):
        shout(message)

shaun = LoudTalker()
shaun.say("Herpa derpa")
```

Classes are declared using the `class` keyword, and methods are declared as functions inside the class body (indicated by indentation). Instances are created by calling the class object, there is no `new` keyword.

Iteration

```
beverage = None
for fridge in office:
    if 'Coke' in fridge:
        beverage = fridge.remove('Coke')
        break
if not beverage:
    shout("Where's the @#$%ing Coke?")
```

All containers can be iterated over with `for ... in ...`. The `in` keyword can also be used for existence

checks with `if`.

Iteration - Dictionaries

If *all* containers can be iterated over, what happens when you iterate over a dictionary?

```
for key in mapping:
  print "%s=%s" % (key, mapping[key])

for value in mapping.values():
  print "???=%s" % value

for (k, v) in mapping.items():
  print "%s=%s" % (k, v)
```

Splats

Functions can declare that they accept a variable number of arguments or arbitrary keyword arguments or both. This is done by prefixing the final argument name with * or ** or having two argument names (with * and ** , respectively):

```
def takes_all(required, *args, **kwargs):
    # tuple of all positional (non-keyword) arguments
    print args

    # dictionary of all keyword arguments
    print kwargs
```

The End

There are some very important keywords that got left out here, but at this point you know enough to write some code and start filling in the gaps.

Questions?

Code Organization

Python organizes code using *modules* and *packages*.

Generally, modules correspond to files and packages correspond to folders.

You access code and data from different modules and packages by using `import` statements.

http://docs.python.org/tutorial/modules.html

Modules

A *module* is any python source file on your python library path. A file named russia.py will correspond to a module named `russia`, and you would write `import russia` to import the `russia` module into your current scope.

Everything declared in the top level of russia.py would be accessible as an attribute of the module like `russia.COASTLINE_LENGTH`

Packages

A *package* is used for the organization of modules. Any folder on your python path that contains a file name __init__.py is a package, and can contain modules or more packages (sub-folders).

Packages and their sub-packages are dot-separated in Python code, so if we moved russia.py into a folder named "country", our code would then become:

```
import country.russia
import country.liechtenstein

assert country.russia.SIZE > country.liechtenstein.SIZE
```

Partial Import

When you have a number of nested sub packages, it's often inconvenient to use names like `country.russia.industry.EXPORTS`. Instead you can import certain parts of Russia's industry directly into your current scope:

```
from country.russia.industry import EXPORTS
assert 'spices' in EXPORTS
```

```
from country.russia import SIZE as size_of_russia
```

Relative Import

Finally, if you have many nested packages, it's also nice to not have to use the full names in your import statement. For example, we could place the following in country/russia/__init__.py:

```
from .industry import EXPORTS
```

Functional Programming

While it may seem odd (or just annoying) to talk about the functional features before the object oriented ones, I think they are simpler to show before rather than after.

Functions are first-class

Functions in Python can be referred to by name. They are run only when you *call* them with parenthesis:

```
def add(x,y): return x + y

def subtract(x,y): return x - y

def do_binary_op(op, x, y):
    return op(x, y)

z = do_binary_op(add, 5, 10) # z = 15
```

Closures

```
def make_rotater(seq):
    def rotate():
        val = seq.pop(0)
        seq.append(val)
        return val
    return rotate

r = make_rotater([1,2,3])
assert [r(), r(), r(), r(), r()] == [1,2,3,1,2]
```

You can safely refer to objects from the scope a function was declared in, even after that scope ends.

Closure Limitation/Gotcha

An annoying implementation detail of Pythons closures is that *only* objects are properly closed over. Non-reference types (strings and numbers) only exist for the duration of the scope they were declared in.

Decorators

Python has a special syntax that allows you to wrap or "decorate" functions (and classes) at compile time:

```
def off_by_one(original_function):
    def new_function(x, y):
        return original_function(x, y) + 1
    return new_function

@off_by_one
def add(x, y):
    return x + y
```

Decorators: What just happened?

The `off_by_one` function takes a function as it's sole argument and returns a new function.

We *decorated* the `add` function with `off_by_one` by placing `@off_by_one` above the function definition. This is equivalent to writing the following after the definition:

```
add = off_by_one(add)
```

http://docs.python.org/glossary.html#term-decorator

Lambda (anonymous) functions

Short functions can be declared inline using the `lambda` syntax:

```
is_odd = lambda x: x % 2
```

These are useful when used with functions that take a function as a parameter, like `sorted`:

```
numbers = [55, 22, 53, 16, 67, 363612, 64361, 12556]
# Sort the numbers by their least significant byte
lsB_ordered = sorted(numbers, key=lambda x: x & 0xFF)
```

http://docs.python.org/library/functions.html#sorted

Generator Functions

Generator functions maintain state between calls. By using `yield` instead of return, the function will pick up on the next line the next time you call it.

```
def get_all_records(lookup, keys):
    for key in keys:
        yield datasource.get(key)

for record in get_all_records(lookup, keys):
    print record
```

Object Oriented Features

Python has first class support for OO programming, including advanced techniques such as abstract base classes, multiple inheritance, and traits/mixins.

http://docs.python.org/tutorial/classes.html

Simple Classes

We earlier saw the syntax for creating a new class:

```
class LoudTalker(object):
    def say(self, message):
        print "%s!" % message

shouter = LoudTalker()
shouter.say("Hi") # Prints "Hi!"
```

Classes Attributes

Our class is really kind of limited, so let's refactor it a bit:

```
class LoudTalker(object):
    suffix = "!"

    def say(self, message):
        print "%s%s" % (message, self.suffix)
```

Here we've added a class attribute `suffix`. This can be referenced either via the class object (`LoudTalker.suffix`) or any class instance (`self.suffix`).

Classes Attributes and Inheritance

It will also be available via any subclass of `LoudTalker`:

```
class SubLoudTalker(LoudTalker):
    pass

assert SubLoudTalker.suffix == "!"

class UnsureTalker(LoudTalker):
    suffix = "..."
```

```
pensively = UnsureTalker()
pensively.say("I'm pretty sure")
```

__init__

Python constructors (actually initializers) are named `__init__`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

stephen = Person('Stephen', 27)
```

What are constructors?

The distinction between constructors and initializers is necessary because there is *also* a constructor method (named `__new__`) that is responsible for creating the backing datastructure of the instance. It is rarely used outside of metaclasses.

super(...)

Sub classes can call parent implementations of methods they've overridden using `super`. The first argument is the type to "skip" in the hierarchy, the second is the object instance.

```
class DoubleTalker(LoudTalker):
    def say(self, message):
        super(DoubleTalker, self).say(message)
        super(DoubleTalker, self).say(message)
```

Evolving APIs with the @property decorator

There is an annoyance when creating "record" style objects in most languages, in that using public properties, while simpler and cleaner, leads to a brittle interface for your class.

What if a property name needs to change? Or some goofball writes `time.minute = 1984`?

Python neatly sidesteps these issues with it's @property decorator...

@property decorator example

```
class Person(object):
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    @property
    def age(self): return self._age

    @age.setter
    def age(self, age):
        assert age >= 0
        self._age = age
```

Abstract Base Classes - Pt. 1

```
import abc # Abstract Base Classes

class AbstractTalker(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def format(self, message):
        return message

    def say(self, message):
        print self.format(message)
```

Abstract Base Classes - Pt. 2

```
class LoudTalker(AbstractTalker)
    def format(self, message):
        return "%s!" % message

class Screamer(LoudTalker):
    def format(self, message):
        return super(Screamer, self).format(message).upper()
```

An abstract base class cannot be instantiated, and subclasses must implement all methods decorated as abstract. However, you can provide an implementation for abstract methods, which subclasses can use via `super`. See http://docs.python.org/library/abc.html for more details.

Mixins/Multiple Inheritance

```
class ShoutFormatterMixin(object):
    def format(self, message):
      return "%s!" % message

class PublicAddressSystem(ShoutFormatterMixin, AbstractTalker):
    def play_music(self, song):
        super(PublicAddressSystem, self).say(song.tablature)
```

Protocols

or: The Magicians Secrets Revealed

What is a "protocol"?

Protocols are similar to interfaces, in that they define a collection of methods an object must support to implement that protocol, but different in that they have language-level recognition and syntactic support. A great example is the iteration protocol...

Iteration

```
for item in some_collection:
    print item
```

How does the above code work for all collection types?

The secret is the iteration protocol. As long as `some_collection` implements a couple of methods it can be used as an iterator.

http://docs.python.org/library/stdtypes.html#iterator-types

Iteration - Example

```
def reverse_iter(seq):
    position = len(seq) - 1
    while True:
        if position < 0:
            raise StopIteration
        yield seq[position]
        position -= 1

class BackwardsSequence(list):
    def __iter__(self):
        return reverse_iter(self)
```

Other Protocols

Here are a few of the many protocols Python supports for customizing your objects interaction with the language:

Comparison (__eq__, __gt__, __lt__)
Containers (__contains__, __setitem__, __getitem__)
Iterators (__iter__, next)
Context Managers (__enter__, __exit__)
Stringification (__str__, __unicode__, __repr__)
Descriptors (__get__, __set__)
Instance Creation (__new__, __metaclass__ attribute)

Container Protocol

By implementing the appropriate methods, it's simple to make your class act like one of the built in container types.

The following example creates a set of "proxy" classes that wrap a Cassandra library to support indexed lookup:

```
keyspace['column_family']['rowkey']['column_name']
```

Container - Example (1/3)

```
from pycassa import *

class KeyspaceProxy(object):
    def __init__(self, keyspace, connection_pool):
        self._keyspace = keyspace
        self._pool = connection_pool

    def __getitem__(self, cf):
        return ColumnFamilyProxy(self._pool, cf)
```

Container - Example (2/3)

```
class ColumnFamilyProxy(object):
    def __init__(self, pool, cf_name):
        self._cf = ColumnFamily(pool, cf_name)

    def __getitem__(self, rowkey):
        return RowProxy(self._cf, rowkey)
```

Container - Example (3/3)

```
class RowProxy(object):
    def __init__(self, column_family, rowkey):
        self._cf = column_family
        self._rowkey = rowkey

    def __getitem__(self, column_name):
```

```
        row = self._cf.get(self._rowkey, [column_name])
        return row[column_name]

    def __setitem__(self, column_name, value):
        self._cf.insert(self._rowkey,
                        {column_name: value})
```

Attribute Access

Another commonly used special method is `__getattr__`. It is called when `your_obj.attr_name` is accessed and does *not* exist, (i.e. right before an AttributeError is raised).

```
class Mock(object):
    def __getattr__(self, attr_name):
        setattr(self, attr_name) = Mock()
        return self.attr_name
```

This is a simplification of an actual mocking library I use called mock.

Descriptors

Descriptors are the mechanism by which method and attribute binding happens in Python. When you perform an attribute access on an object (e.g. `object.attribute`) Python gives `attribute` an opportunity to customize it's behaviour at access time, **if** it implements the descriptor protocol.

The `object.attribute` call gets transformed into something like `attribute.__get__(object)` and the return value of `__get__` is what appears to be `object.attribute` to the calling code.

Descriptor - Example

```
# Attach a new function to an existing object

class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

me = Person("Stephen", 27)
```

Descriptor - Example

```
def sleep(self):
    if self.age <= 25:
        print "later"
    else:
```

```
        print "zzz"

# Bind the sleep method to myself
me.sleep = sleep.__get__(me)
me.sleep() # Prints "zzz"
```

The End

Python protocols allow you to take full advantage of the complete syntax of the language in your own objects. Writing library code against simple built-in types that can be used with any type that supports the operations needed is a great example of *duck-typing* in Python: if it walks like a duck and talks like a duck, code that only needs a duck shouldn't care that it's a subclass of `DeadlyRobot` with a `DuckMixin`.