Concise Python Cheat Sheet

NOTE: this is a work in progress

Structure

```
#!/usr/bin/env python

def hello(who):
    print("Hello " + who + "!");

hello("world")
```

Standard Classes and Types

```
NoneType   None
ellipsis   ...
bool       False
int        5040
float      3.141
str        "Escape \"double-quotes\", not 'single'"
str        'Escape \'single-quotes\', not "double"'
list       [360,'abc',3.141,360]
set        {1,'abc',3}
dict       {'abc': 10, 2: 3, 7: 'def'}
```

Evaluation Order

Expressions are evaluated left-to-right:

```
1st + (2nd * 3rd)
(1st + 2nd) * 3rd
3rd, 4th = 1st, 2nd
```

Modules

qualified module import	`import module` `module.foo()`
unqualified module import	`from module import *` `foo()`
import specific name	`from module import foo` `foo()`
import module renaming	`import module as m` `m.foo()`
import specific rename	`from module import foo as f` `f()`

Operators (grouped by precedence)

tuples, lists, dictionaries	`(...), [...], {...}`
attribute reference	`.`
function call, indexing&slicing	`name(args), name[idx]`
exponentiation	`**`
bitwise negation	`~`
unary identity, unary negation	`-, +`

multiplication&repetition, division
integer division, integer remainder
matrix multiplication
	`*, /`
	`//, %`
	`@`

addition & concat, subtraction	`+, -`
left shift, right shift	`<<, >>`
bitwise and	`&`
bitwise xor	`^`
bitwise or	`|`
comparisons (chaining) object identity, set membership	`==, !=, <, <=, >, >=` `is, is not, in, not in`
logical negation	`not`
conjunction	`and`
disjunction	`or`
ternary selection	`expr1 if cond else expr2`
lambda	`lambda args: expr`

Functions

referencing	`fun`
calling	`fun()`
calling (w/arg)	`fun(arg_value)`
calling (w/kw arg)	`fun(1st_arg, 2nd_arg, key=val)`
0-arguments	`def fun():` `  code-block`
1-argument	`def fun(arg):` `  code-block`
2-arguments	`def fun(arg0,arg1):` `  code-block`
default args	`def fun(mandatory,optional=default):` `  code-block`
arbitrary args	`def fun(mandatory, *args):` `  code-block`
lambda (1-arg)	`lambda arg: expression`
lambda (2-args)	`lambda arg0,arg1: expression`

docstring	`def fun(args):`
	`  """Concise summary of purpose`
	`  Longer description, ifneedbe`
	`  """`
	`  code-block`

Flow Control

if statement	`if condition:` `  code-block`

if-else statement	`if condition1:`
	`  code-block`
	`else:`
	`  code-block`

if-elif-else statement	`if condition1:`
	`  code-block`
	`elif condition2:`
	`  code-block`
	`else:`
	`  code-block`

while statement	`while condition:` `  code-block`
for statement	`for v in values:` `  code-block`
for statement (copy)	`for v in values[:]:` `  code-block`
for statement (count)	`for i in range(n):` `  code-block  # from 0 to n`

break statement	`while-or-for-statement:`
	`  break  # exit loop, skip else`
	`else:`
	`  code-block`

continue-statement	`while-or-for-statement:` `  continue  # next loop iteraction`
pass-statement (no-op)	`def do-nothing():` `  pass`
case/switch statements	There aren't any. Use if-elif-else.

List Comprehensions

Take *element* from *list*. If *boolPredicate*, add element *expr* to list:

```
[expr for element in list if boolPredicate ...]
```

```
[x for x in xs]             ≡ xs
[f(x) for x in [a,b,c]]     ≡ [f(a),f(b),f(c)]
[f(x) for x in xs if p(x)] ≡ list(map(f,filter(p,xs)))
[x+y for x in [a,b] for y in [i,j]]
                                ≡ [a+i, a+j, b+i, b+j]
```

Generator `(...)` and set `{...}` comprehensions are also allowed.

Exceptions

try-except
```
try:
    code-block
except Exception:
    code-block
```

finally-else
```
try:
    code-block
except Exception1:
    run when Exception1 is raised
except Exception2:
    run when Exception2 is raised
else:
    run when no exception is raised
finally:
    always-run
```

catch BaseException	`except:  # dangerous`
catch all Exceptions	`except Exception:`
catch as	`except Ex1 as ex:`
catch multiple exceptions	`except (Ex1, Ex2, Ex3):`
exception arguments (as)	`ex.args`
raise exception *Ex1*	`raise Ex1`
raise *Ex1* with "msg"	`raise Ex1("msg")`
re-raise (inside except:)	`raise`

Exception Hierarchy

BaseException	not to be directly inherited
SystemExit	raised by sys.exit()
KeyboardInterrupt	raised on interrupt / ctrl-C
Exception	derive **user-defined exceptions**
StopIteration	no further items on iterator
ArithmeticError	arithmetic error
ZeroDivisionError	raised on division by zero
AssertionError	raised by assert()
AttributeError	inaccessible/nonexistent attribute
BufferError	error on a buffer operation
EOFError	raised by input() on EOF
LookupError	error on lookup
IndexError	out of range list index
KeyError	key not found on dictionary
NameError	name not found
OSError	raised by the OS
FileExistsError	file exists
FileNotFoundError	file not found
PermissionError	inadequate access rights
RuntimeError	runtime error, see error string
NotImplementedError	unimplemented abstract method
RecursionError	maximum recursion depth reached
TypeError	inappropriate type
ValueError	right type, innapropriate value
Warning	warnings

Functions

Objects

list attributes & methods of object *o*	`dir(o)`
shows the help for object *o*	`help(o)`
type/class of *x*	`type(x)`
base classes of *cls*	`cls.__bases__`

Lists

append x to the end of list xs	`xs.append(x)`
append iterable iter to the end of list xs	`xs.extend(iter)`
insert x in position i on the list xs	`xs.insert(i, x)`
remove the first occurrence of x in xs	`xs.remove(x)`
remove then return the last element of xs	`xs.pop()`
remove then return the i-th element of xs	`xs.pop(i)`
remove all items from xs	`xs.clear()`
index of the first occurrence of x	`xs.index(x)`
between i and j	`xs.index(x,i,j)`
count occurrences of x in xs	`xs.count(x)`
sort the list xs	`xs.sort()`
sort the list xs using keying function k	`xs.sort(key=k)`
reverse the list xs	`xs.reverse()`
return a copy of the list xs	`xs.copy()`

Printing

printable representation of *x*	`str(x)`
parsable printable representation of *x*	`repr(x)`
identity of *x*	`parse(repr(x))`

String formatting

format *x* to string using str	`"{0}".format(x)`
format *x* to decimal	`"{0:d}".format(x)`
format *x* to hexadecimal	`"{0:x}".format(x)`
format *x* to octal	`"{0:o}".format(x)`
replace {0} by *s*	`"... {0} ...".format(s)`
repl. {0} and {1} by *s0* and *s1*	`". {0} . {1} ." % (s0,s1)`
generate "pi is 3.14"	`"pi is {0:2f}".format(math.pi)`
generate "pi is 003.141"	`"pi is {0:3.3f}".format(math.pi)`

Old printf-style string formatting

format *x* to string using str	`"%s" % x`
format *x* to string using repr	`"%s" % x`
format *x* to decimal	`"%d" % x`
format *x* to hexadecimal	`"%x" % x`
format *x* to octal	`"%o" % x`
replace %s by *s*	`"... %s ..." % s`
repl. %s by *s1* then *s2*	`". %s . %s ." % (s1,s2)`
generate "pi is 3.14"	`"pi is %.2f" % math.pi`
generate "pi is 003.141"	`"pi is %3.3f" % math.pi`
generate "1 + 1 = 2"	`"1 + 1 = %d" % (1 + 1)`

Input and output to files

open a file for reading	`f = open('file.txt')`
open a file for reading	`f = open('file.txt','r')`
open a file for writing	`f = open('file.txt','w')`
open a file for writing (appending)	`f = open('file.txt','a')`
closes file *f*	`close(f)`
opens and closes	`with open('file.txt') as f:` `  operations on f`
reads the entire contents of file *f*	`f.read()`
reads a line from file *f*	`f.readline()`
reads all lines from file *f*	`f.readlines()`
loop over all lines from file *f*	`for line in f:` `  code-block`
writes *string* on file *f*	`f.write(string)`

Classes

defining a class
```
class OurClass:
    """Description of OurClass"""
    class_var = val   # shared

    def f(self):
        self.instance_var = val
        code
```

inheritance	`class OurSubClass(OurClass):`
multiple inheritance	`class SubClass(Cl1,Cl2):`
object instance	`x = MyClass()`
attribute reference	`x.x`
class var reference	`OurClass.class_var`
class var reference on obj	`x.class_var`
instance var reference	`x.instance_var`
method call	`x.f()`

initializer/constructor
```
def __init__(self, arg1, arg2):
    code
```

simple record
```
class Record:
    pass

record = Record()
record.x = value1
record.y = value2
```

Iterators & Generators

declaring
```
def generator()
    yield value1
    yield value2
    yield value3
```

using
```
for value in generator():
    code-block
```