Python cheat sheet

Code structure

Grouping: Whitespace has meaning. Line breaks separate statements, indentation creates logical blocks. Comments run from # to line break. Functional units go into modules (files) and packages (directories); each source file imports any modules it uses:

```python
import math
for x in range(0, (4+1)*2):  # numbers 0 <= x < 10
    y = math.sqrt(x)
    print('The square root of {} is {}'.format(
        x, y))
```

Variable names: May contain letters (unicode, case-sensitive), numerals and _.

Logic and flow control

Conditions: compound statement or expression:

```python
if x < y:                print('equal'
    print(x)                 if x == y
elif x > y:                  else 'unequal')
    print(y)
else:
    print('equal')
```

Iteration: over sets or until termination:

```python
for name in ['John', 'Fred', 'Bob']:
    if name.startswith('F'):
        continue
    print(name)


while input('Stop?') != 'stop':
    if 'x' in input('Do not type an x.'):
        print('You typed an x.')
        break
else:
    print('Loop finished without typing an x.')
```

Set and mapping types (unordered):

```python
set         {'Fred', 'John'}  set(('Fred', 'John'))
frozenset   frozenset(('Fred', 'John'))
dict        {'Fred': 123, 42: 'John'}
            dict([('Fred', 123), (42, 'John')])
            dict(Fred=123, John=42)
```

Immutable set methods and operations:

```python
intersection()  symmetric_difference()  issubset()
union()         copy()                  issuperset()
difference()                            isdisjoint()
```

```python
{1, 2} & {2, 3} == {2}         {1, 2} == {2, 1}
{1, 2} | {2, 3} == {1, 2, 3}        {1} < {1, 2}
{1, 2} - {2, 3} == {1}         {1, 2} <= {1, 2}
{1, 2} ^ {2, 3} == {1, 3}
```

Set mutation methods:

```python
add()     update()    intersection_update()
pop()     remove()    difference_update()
clear()   discard()   symmetric_difference_update()
```

Mapping methods and operations:

```python
get()       keys()      pop()       copy()
setdefault() values()   popitem()   fromkeys()
update()    items()     clear()
```

```python
x = {'a': 1, 'b': 2};   x['d'] = 5
'b' in x == True;       x['a'] == 1;   del x['b']
```

List and dict comprehensions:

```python
[2 * i for i in range(3)] == [0, 2, 4]
{i: i ** 2 for i in range(3)}
        == {0: 0, 1: 1, 2: 4}
```

Functions

Simple function definition, takes an argument of any type:

```python
def double(x):              double(2) == 4
    return x * 2            double('abc') == 'abcabc'
```

Function that does not explicitly return a value:

```python
def idle(): pass        idle() == None
```

Function with optional arguments:

```python
def multiply(x, y=2):            multiply(3) == 6
    return x * y                multiply(3, 5) == 15
                                multiply(3, y=5) == 15
```

Classes

Simple class definition with attributes and constructor:

```python
class Simple:                   obj = Simple(7)
    x = None                    obj.x == 7
    def __init__(self, x):
        self.x = x
```

Subclass which accesses a method of its Superclass:

```python
class XY(Simple):               obj = XY(7, 9)
    y = None                    obj.x == 7
    def __init__(self, x, y):   obj.y == 9
        super().__init__(x)
        self.y = y
```

Class with a method that can be called on instances:

```python
class CalcZ(XY):                obj = CalcZ(7, 9)
    def do_z(self):             obj.do_z() == 63
        return self.x * self.y
```

Class with an automatically computed attribute:

```python
class AutoZ(XY):                obj = AutoZ(7, 9)
    @property                   obj.z == 63
    def z(self):
        return self.x * self.y
```

This cheat sheet refers to Python 3.7:
https://docs.python.org/3.7/

Coding style conventions according to PEP8
https://python.org/dev/peps/pep-0008/

Text by Kristian Rother, Thomas Lotze (CC-BY-SA 4.0)

https://www.cusy.io/de/seminare

>CUSY_

Exceptions: for explicit error handling:

```python
try:
    if 'x' in input('Do not type an x.'):
        raise RuntimeError('You typed an x.')
except Exception as exc:
    print(exc)
else:
    print('You did not type an x.')
finally:
    print('Good bye.')
```

Context managers: implicit error handling for resources:

```python
with open('story.txt', 'w') as story:
    print('Once upon a time...', file=story)
```

Built-in functions

Input and output:
```python
input([prompt])          open(file, ...)
print(*objects, file=sys.stdout, ...)
```

Collections:
```python
iter(obj[, sentinel])  next(iterator)
all(iterable)          filter(function, iterable)
any(iterable)          map(function, *iterables)
max(iterable)          reversed(sequence)
min(iterable)          sorted(iterable, ...)
len(sequence)          enumerate(iterable)
sum(iterable[, start]) zip(*iterables)
```

Object representation:
```python
ascii(obj)             format(obj[, format_spec])
repr(obj)
```

Object manipulation and reflection:
```python
dir([obj])             isinstance(obj, classinfo)
vars([obj])            issubclass(class, classinfo)
hasattr(obj, name)     setattr(obj, name, value)
getattr(obj, name)     delattr(obj, name)
```

Data types

Numeric types:
```python
int      137   -42   1_234_567  0b1011  0o177  0x3f
float    2.71  .001  2.718_281  5.43e-10
complex  0.3j  5J    (1 - 2.5j)

int(1)   int('2_345')   int('0xff')   int('  1  ')
float(12)       float('2.71')       float('1.4e9')
complex('5j')  complex(1, -2.3)
str(123.0) == '123.0';  bin(23)  oct(23)  hex(23)
```

Numeric operations:
```python
1 + 1 == 2; 7 / 2 == 3.5; 7 // 2 == 3;  7 % 2 == 1
2 - 1 == 1; 2 * 3 == 6;      divmod(7, 2) == (3, 1)
2 ** 3 == 8;         (1 + 3j).conjugate() == 1 - 3j
pow(2, 3) == 8;      abs(-1) == 1;   round(1.5) == 2
```

Boolean type (truth values):
```python
bool    True    False
bool(123) == True;  bool(0) == False
```

Boolean operations:
```python
True and False == False;      True or False == True
not True == False; not 42 == False;  0 or 42 == 42
```

Text (unicode) strings:
```python
str    'abc'      """abc"""            """some
       "a'b'c"    'a\'b\'c'            multiline
       'äbc'      'a\xfcc'    'ab\nc'  string"""

ord('A') == 65;    chr(65) == 'A'
'äbc'.encode('utf-8') == b'\xc3\xa4bc'
```

String formatting:
```python
'Mr {name}: {age} years old.'.format(
    name='Doe', age=42) == 'Mr Doe: 42 years old.'

name = 'Doe'; age = 42
f'Mr {name}: {age} years' == 'Mr Doe: 42 years'
```

String methods:
```python
upper()     casefold()    title()
lower()     swapcase()    capitalize()

center()    ljust()     rjust()
lstrip()    rstrip()    strip()

count()     index()     rindex()    find()    rfind()

join()      partition()  rpartition()
split()     rsplit()     splitlines()

replace()   format()     translate()  expandtabs()
zfill()     format_map()  maketrans()

isdigit()   isdecimal()  isupper()    startswith()
isalpha()   isnumeric()  islower()    endswith()
isalnum()   isprintable() istitle()
isspace()   isidentifier()
```

Sequence types:
```python
tuple   ()    (1,)   (1, 'abc', 3.4)
list    []    [1]    [1.0, 'abc', [1, 2, 3]]
range   tuple(range(1, 4)) == (1, 2, 3)

list('ab') == ['a', 'b'];  tuple([1, 2]) == (1, 2)
(1, 1, 2).count(1) == 2;   (1, 2, 3).index(3) == 2
```

Sequence and string operations, slicing:
```python
'ab' * 3 == 'ababab'; [1, 2] in [0, 1, 2] == False
'ab' + 'cd' == 'abcd';    'bc' in 'abcd' == True
(1, 2) + (3,) == (1, 2, 3);  1 in (0, 1) == True

'abc'[1] == 'b';          (1, 2, 3)[-1] == 3
'abcd'[1:3] == 'bc';      [1, 2][:] == [1, 2]
'abcd'[1:] == 'bcd';      [1, 2][:] is not [1, 2]
'abcdefgh'[1:7:2] == 'bdf'
```

List mutation methods and operations:
```python
append()    pop()     copy()    sort()    extend()
insert()    remove()  clear()   reverse()

x = [1, 2];   x += [3];   x *= 2;   del x[4]
del x[1:3];   x[:2] = [4, 5, 6]
```