CONTENTS

NUMBERS

Python's 2 main types for Numbers is int and float (or integers and floating point numbers).

```
type(1)   # int
type(-10) # int
type(0)   # int
type(0.0) # float
type(2.2) # float
type(4E2) # float - 4*10 to the power of 2
```

```
# Arithmetic
10 + 3  # 13
10 - 3  # 7
10 * 3  # 30
10 ** 3 # 1000
10 / 3  # 3.3333333333333335
10 // 3 # 3 --> floor division - no decimals and returns an int
10 % 3  # 1 --> modulo operator - return the reminder. Good for deciding if
number is even or odd
```

```
# Basic Functions
pow(5, 2)      # 25 --> like doing 5**2
abs(-50)       # 50
round(5.46)    # 5
round(5.468, 2)# 5.47 --> round to nth digit
bin(512)       # '0b1000000000' -->  binary format
hex(512)       # '0x200' --> hexadecimal format
```

```
# Converting Strings to Numbers
age = input("How old are you?")
age = int(age)
pi = input("What is the value of pi?")
pi = float(pi)
```

STRINGS

Strings in python are stored as sequences of letters in memory.

```
type('Hellloooooo') # str

'I\'m thirsty'
"I'm thirsty"
"\n" # new line
"\t" # adds a tab

'Hey you!'[4] # y
name = 'Andrei Neagoie'
name[4]     # e
name[:]     # Andrei Neagoie
name[1:]    # ndrei Neagoie
name[:1]    # A
name[-1]    # e
name[::1]   # Andrei Neagoie
name[::-1]  # eiogaeN ierdnA
name[0:10:2]# Ade e
# : is called slicing and has the format [ start : end : step ]

'Hi there ' + 'Timmy' # 'Hi there Timmy' --> This is called string concatenation
'*'*10 # **********
```

```
# Basic Functions
len('turtle') # 6

# Basic Methods
'  I am alone '.strip()                # 'I am alone' --> Strips all whitespace
characters from both ends.
'On an island'.strip('d')              # 'On an islan' --> # Strips all passed
characters from both ends.
'but life is good!'.split()            # ['but', 'life', 'is', 'good!']
'Help me'.replace('me', 'you')         # 'Help you' --> Replaces first with
second param
'Need to make fire'.startswith('Need')# True
'and cook rice'.endswith('rice')       # True
'bye bye'.index(e)                     # 2
'still there?'.upper()                 # STILL THERE?
'HELLO?!'.lower()                      # hello?!
'ok, I am done.'.capitalize()          # 'Ok, I am done.'
'oh hi there'.find('i')                # 4 --> returns the starting
index position of the first occurrence
'oh hi there'.count('e')               # 2
```

```
# String Formatting
name1 = 'Andrei'
name2 = 'Sunny'
print(f'Hello there {name1} and {name2}')          # Hello there Andrei and
Sunny - Newer way to do things as of python 3.6
print('Hello there {} and {}'.format(name1, name2))   # Hello there Andrei and
Sunny
print('Hello there %s and %s' %(name1, name2))     # Hello there Andrei and
Sunny --> you can also use %d, %f, %r for integers, floats, string
representations of objects respectively
```

```
#Palindrome check
word = 'reviver'
p = bool(word.find(word[::-1]) + 1)
print(p) # True
```

BOOLEAN

True or False. Used in a lot of comparison and logical operations in Python.

```
bool(True)
bool(False)

# all of the below evaluate to False. Everything else will evaluate to True in Python.
print(bool(None))
print(bool(False))
print(bool(0))
print(bool(0.0))
print(bool([]))
print(bool({}))
print(bool(()))
print(bool(''))
print(bool(range(0)))
print(bool(set()))

# See Logical Operators and Comparison Operators section for more on booleans.
```

LISTS

Unlike strings, lists are mutable sequences in python.

```
my_list = [1, 2, '3', True]# We assume this list won't mutate for each example
below
len(my_list)                # 4
my_list.index('3')          # 2
my_list.count(2)            # 1 --> count how many times 2 appears

my_list[3]                  # True
my_list[1:]                 # [2, '3', True]
my_list[:1]                 # [1]
my_list[-1]                 # True
my_list[::1]                # [1, 2, '3', True]
my_list[::-1]               # [True, '3', 2, 1]
my_list[0:3:2]              # [1, '3']

# : is called slicing and has the format [ start : end : step ]
```

```
# Add to List
my_list * 2                 # [1, 2, '3', True, 1, 2, '3', True]
my_list + [100]             # [1, 2, '3', True, 100] --> doesn't mutate original
list, creates new one
my_list.append(100)         # None --> Mutates original list to [1, 2, '3', True,
100]          # Or: <list> += [<el>]
my_list.extend([100, 200]) # None --> Mutates original list to [1, 2, '3', True,
100, 200]
my_list.insert(2, '!!!')   # None -->  [1, 2, '!!!', '3', True] - Inserts item
at index and moves the rest to the right.

' '.join(['Hello','There'])# 'Hello There' --> Joins elements using string as
separator.
```

```
# Copy a List
basket = ['apples', 'pears', 'oranges']
new_basket = basket.copy()
new_basket2 = basket[:]
```

```
# Remove from List
[1,2,3].pop()    # 3 --> mutates original list, default index in the pop method
is -1 (the last item)
[1,2,3].pop(1)   # 2 --> mutates original list
[1,2,3].remove(2)# None --> [1,3] Removes first occurrence of item or raises
ValueError.
[1,2,3].clear()  # None --> mutates original list and removes all items: []
del [1,2,3][0] #
```

```
# Ordering
[1,2,5,3].sort()          # None --> Mutates list to [1, 2, 3, 5]
[1,2,5,3].sort(reverse=True) # None --> Mutates list to [5, 3, 2, 1]
[1,2,5,3].reverse()       # None --> Mutates list to [3, 5, 2, 1]
sorted([1,2,5,3])         # [1, 2, 3, 5] --> new list created
list(reversed([1,2,5,3]))# [3, 5, 2, 1] --> reversed() returns an iterator
```

```
# Useful operations
1 in [1,2,5,3]  # True
min([1,2,3,4,5])# 1
max([1,2,3,4,5])# 5
sum([1,2,3,4,5])# 15
```

```
# Get First and Last element of a list
mList = [63, 21, 30, 14, 35, 26, 77, 18, 49, 10]
first, *x, last = mList
print(first) #63
print(last) #10
```

```
# Matrix
matrix = [[1,2,3], [4,5,6], [7,8,9]]
matrix[2][0] # 7 --> Grab first first of the third item in the matrix object

# Looping through a matrix by rows:
mx = [[1,2,3],[4,5,6]]
for row in range(len(mx)):
      for col in range(len(mx[0])):
            print(mx[row][col]) # 1 2 3 4 5 6

# Transform into a list:
[mx[row][col] for row in range(len(mx)) for col in range(len(mx[0]))] #
[1,2,3,4,5,6]

# Combine columns with zip and *:
```

```
# List Comprehensions
# new_list[<action> for <item> in <iterator> if <some condition>]
a = [i for i in 'hello']                  # ['h', 'e', 'l', 'l', '0']
b = [i*2 for i in [1,2,3]]                # [2, 4, 6]
c = [i for i in range(0,10) if i % 2 == 0]# [0, 2, 4, 6, 8]
```

```
# Advanced Functions
list_of_chars = list('Helloooo')                                    # ['H', 'e',
'l', 'l', 'o', 'o', 'o', 'o']
sum_of_elements = sum([1,2,3,4,5])                                  # 15
element_sum = [sum(pair) for pair in zip([1,2,3],[4,5,6])]          # [5, 7, 9]
sorted_by_second = sorted(['hi','you','man'], key=lambda el: el[1])# ['man',
'hi', 'you']
sorted_by_key = sorted([
                        {'name': 'Bina', 'age': 30},
                        {'name':'Andy', 'age': 18},
                        {'name': 'Zoey', 'age': 55}],
                        key=lambda el: (el['name']))# [{'name': 'Andy', 'age':
18}, {'name': 'Bina', 'age': 30}, {'name': 'Zoey', 'age': 55}]
```

```
# Read line of a file into a list
with open("myfile.txt") as f:
  lines = [line.strip() for line in f]
```

DICTIONARIES

Also known as mappings or hash tables. They are key value pairs that are guaranteed to retain order of insertion starting from Python 3.7.

```
my_dict = {'name': 'Andrei Neagoie', 'age': 30, 'magic_power': False}
my_dict['name']                         # Andrei Neagoie
len(my_dict)                            # 3
list(my_dict.keys())                    # ['name', 'age', 'magic_power']
list(my_dict.values())                  # ['Andrei Neagoie', 30, False]
list(my_dict.items())                   # [('name', 'Andrei Neagoie'), ('age', 30),
('magic_power', False)]
my_dict['favourite_snack'] = 'Grapes'# {'name': 'Andrei Neagoie', 'age': 30,
'magic_power': False, 'favourite_snack': 'Grapes'}
my_dict.get('age')                      # 30 --> Returns None if key does not
exist.
my_dict.get('ages', 0 )                 # 0 --> Returns default (2nd param) if key
is not found

#Remove key
del my_dict['name']
my_dict.pop('name', None)
```

```
my_dict.update({'cool': True})                                    # {'name':
'Andrei Neagoie', 'age': 30, 'magic_power': False, 'favourite_snack': 'Grapes',
'cool': True}
{**my_dict, **{'cool': True} }                                    # {'name':
'Andrei Neagoie', 'age': 30, 'magic_power': False, 'favourite_snack': 'Grapes',
'cool': True}
new_dict = dict([['name','Andrei'],['age',32],['magic_power',False]])  # Creates
a dict from collection of key-value pairs.
new_dict = dict(zip(['name','age','magic_power'],['Andrei',32, False]))# Creates
a dict from two collections.
new_dict = my_dict.pop('favourite_snack')                         # Removes
item from dictionary.
```

```
# Dictionary Comprehension
{key: value for key, value in new_dict.items() if key == 'age' or key == 'name'}
# {'name': 'Andrei', 'age': 32} --> Filter dict by keys
```

TUPLES

Like lists, but they are used for immutable things (that don't change).

```
my_tuple = ('apple','grapes','mango', 'grapes')
apple, grapes, mango, grapes = my_tuple# Tuple unpacking
len(my_tuple)                          # 4
my_tuple[2]                            # mango
my_tuple[-1]                           # 'grapes'
```

```
# Immutability
my_tuple[1] = 'donuts'  # TypeError
my_tuple.append('candy')# AttributeError
```

```
# Methods
my_tuple.index('grapes') # 1
my_tuple.count('grapes') # 2
```

```
# Zip
list(zip([1,2,3], [4,5,6])) # [(1, 4), (2, 5), (3, 6)]
```

```
# unzip
z = [(1, 2), (3, 4), (5, 6), (7, 8)] # Some output of zip() function
unzip = lambda z: list(zip(*z))
unzip(z)
```

SETS

Unordered collection of unique elements.

```
my_set = set()
my_set.add(1)  # {1}
my_set.add(100)# {1, 100}
my_set.add(100)# {1, 100} --> no duplicates!
```

```
new_list = [1,2,3,3,3,4,4,5,6,1]
set(new_list)           # {1, 2, 3, 4, 5, 6}

my_set.remove(100)      # {1} --> Raises KeyError if element not found
my_set.discard(100)     # {1} --> Doesn't raise an error if element not found
my_set.clear()          # {}
new_set = {1,2,3}.copy()# {1,2,3}
```

```
set1 = {1,2,3}
set2 = {3,4,5}
set3 = set1.union(set2)                 # {1,2,3,4,5}
set4 = set1.intersection(set2)          # {3}
set5 = set1.difference(set2)            # {1, 2}
set6 = set1.symmetric_difference(set2)# {1, 2, 4, 5}
set1.issubset(set2)                     # False
set1.issuperset(set2)                   # False
set1.isdisjoint(set2)                   # False --> return True if two sets have a
null intersection.
```

```
# Frozenset
# hashable --> it can be used as a key in a dictionary or as an element in a
set.
<frozenset> = frozenset(<collection>)
```

NONE

None is used for absence of a value and can be used to show nothing has been assigned to an object.

```
type(None) # NoneType
a = None
```

COMPARISON OPERATORS

```
==                    # equal values
!=                    # not equal
>                     # left operand is greater than right operand
<                     # left operand is less than right operand
>=                    # left operand is greater than or equal to right operand
<=                    # left operand is less than or equal to right operand
<element> is <element> # check if two operands refer to same object in memory
```

LOGICAL OPERATORS

```
1 < 2 and 4 > 1 # True
1 > 3 or 4 > 1  # True
1 is not 4      # True
not True        # False
1 not in [2,3,4]# True

if <condition that evaluates to boolean>:
  # perform action1
elif <condition that evaluates to boolean>:
  # perform action2
else:
  # perform action3
```

LOOPS

```
my_list = [1,2,3]
my_tuple = (1,2,3)
my_list2 = [(1,2), (3,4), (5,6)]
my_dict = {'a': 1, 'b': 2. 'c': 3}

for num in my_list:
    print(num) # 1, 2, 3

for num in my_tuple:
    print(num) # 1, 2, 3

for num in my_list2:
    print(num) # (1,2), (3,4), (5,6)

for num in '123':
    print(num) # 1, 2, 3

for k,v in my_dict.items(): # Dictionary Unpacking
    print(k) # 'a', 'b', 'c'
    print(v) # 1, 2, 3

while <condition that evaluates to boolean>:
  # action
  if <condition that evaluates to boolean>:
    break # break out of while loop
  if <condition that evaluates to boolean>:
    continue # continue to the next line in the block
```

```
# waiting until user quits
msg = ''
while msg != 'quit':
    msg = input("What should I do?")
    print(msg)
```

RANGE

```
range(10)           # range(0, 10) --> 0 to 9
range(1,10)         # range(1, 10)
list(range(0,10,2))# [0, 2, 4, 6, 8]
```

ENUMERATE

```
for i, el in enumerate('helloo'):
  print(f'{i}, {el}')
# 0, h
# 1, e
# 2, l
# 3, l
# 4, o
# 5, o
```

COUNTER

```
from collections import Counter
colors = ['red', 'blue', 'yellow', 'blue', 'red', 'blue']
counter = Counter(colors)# Counter({'blue': 3, 'red': 2, 'yellow': 1})
counter.most_common()[0] # ('blue', 3)
```

NAMED TUPLE

- **Tuple is an immutable and hashable list.**
- **Named tuple is its subclass with named elements.**

```
from collections import namedtuple
Point = namedtuple('Point', 'x y')
p = Point(1, y=2)# Point(x=1, y=2)
p[0]             # 1
p.x              # 1
getattr(p, 'y')  # 2
p._fields        # Or: Point._fields #('x', 'y')
```

```
from collections import namedtuple
Person = namedtuple('Person', 'name height')
person = Person('Jean-Luc', 187)
f'{person.height}'           # '187'
'{p.height}'.format(p=person)# '187'
```

ORDEREDDICT

Maintains order of insertion.

```
from collections import OrderedDict
# Store each person's languages, keeping # track of who responded first.
programmers = OrderedDict()
programmers['Tim'] = ['python', 'javascript']
programmers['Sarah'] = ['C++']
programmers['Bia'] = ['Ruby', 'Python', 'Go']

for name, langs in programmers.items():
    print(name + '-->')
    for lang in langs:
      print('\t' + lang)
```

FUNCTIONS

***args and **kwargs**

Splat (*) expands a collection into positional arguments, while splatty-splat (**) expands a dictionary into keyword arguments.

```
args   = (1, 2)
kwargs = {'x': 3, 'y': 4, 'z': 5}
some_func(*args, **kwargs) # same as some_func(1, 2, x=3, y=4, z=5)
```

*** Inside Function Definition**

Splat combines zero or more positional arguments into a tuple, while splatty-splat combines zero or more keyword arguments into a dictionary.

```
def add(*a):
    return sum(a)

add(1, 2, 3) # 6
```

Ordering of parameters:

```
def f(*args):                   # f(1, 2, 3)
def f(x, *args):                # f(1, 2, 3)
def f(*args, z):                # f(1, 2, z=3)
def f(x, *args, z):             # f(1, 2, z=3)

def f(**kwargs):                # f(x=1, y=2, z=3)
def f(x, **kwargs):             # f(x=1, y=2, z=3) | f(1, y=2, z=3)

def f(*args, **kwargs):         # f(x=1, y=2, z=3) | f(1, y=2, z=3) | f(1, 2,
z=3) | f(1, 2, 3)
def f(x, *args, **kwargs):      # f(x=1, y=2, z=3) | f(1, y=2, z=3) | f(1, 2,
z=3) | f(1, 2, 3)
def f(*args, y, **kwargs):      # f(x=1, y=2, z=3) | f(1, y=2, z=3)
```

Other Uses of *

```
[*[1,2,3], *[4]]                # [1, 2, 3, 4]
{*[1,2,3], *[4]}                # {1, 2, 3, 4}
(*[1,2,3], *[4])                # (1, 2, 3, 4)
{**{'a': 1, 'b': 2}, **{'c': 3}}# {'a': 1, 'b': 2, 'c': 3}
```

```
head, *body, tail = [1,2,3,4,5]
```

LAMBDA

```
# lambda: <return_value>
# lambda <argument1>, <argument2>: <return_value>
```

```
# Factorial
from functools import reduce
n = 3
```

```
# Fibonacci
fib = lambda n : n if n <= 1 else fib(n-1) + fib(n-2)
result = fib(10)
print(result) #55
```

COMPREHENSIONS

```
<list> = [i+1 for i in range(10)]         # [1, 2, ..., 10]
<set>  = {i for i in range(10) if i > 5}  # {6, 7, 8, 9}
<iter> = (i+5 for i in range(10))         # (5, 6, ..., 14)
<dict> = {i: i*2 for i in range(10)}      # {0: 0, 1: 2, ..., 9: 18}
```

```
output = [i+j for i in range(3) for j in range(3)] # [0, 1, 2, 1, 2, 3, 2, 3, 4]

# Is the same as:
output = []
for i in range(3):
  for j in range(3):
    output.append(i+j)
```

TERNARY CONDITION

```
# <expression_if_true> if <condition> else <expression_if_false>

[a if a else 'zero' for a in [0, 1, 0, 3]] # ['zero', 1, 'zero', 3]
```

MAP FILTER REDUCE

```
from functools import reduce
list(map(lambda x: x + 1, range(10)))           # [1, 2, 3, 4, 5, 6, 7, 8,
9,10]
list(filter(lambda x: x > 5, range(10)))        # (6, 7, 8, 9)
list(reduce(lambda acc, x: acc + x, range(10)))  # 45
```

ANY ALL

```
any([False, True, False])# True if at least one item in collection is truthy,
False if empty.
all([True,1,3,True])     # True if all items in collection are true
```

CLOSURES

We have a closure in Python when:

- **A nested function references a value of its enclosing function and then**
- **The enclosing function returns the nested function.**

```
def get_multiplier(a):
    def out(b):
        return a * b
    return out
```

```
>>> multiply_by_3 = get_multiplier(3)
>>> multiply_by_3(10)
30
```

- **If multiple nested functions within enclosing function reference the same value, that value gets shared.**
- **To dynamically access function's first free variable use `'<function>.__closure__[0].cell_contents'`.**

SCOPE

If variable is being assigned to anywhere in the scope, it is regarded as a local variable, unless it is declared as a 'global' or a 'nonlocal'.

```
def get_counter():
    i = 0
    def out():
        nonlocal i
        i += 1
        return i
    return out
```

```
>>> counter = get_counter()
>>> counter(), counter(), counter()
(1, 2, 3)
```

MODULES

```
if __name__ == '__main__': # Runs main() if file wasn't imported.
    main()
```

```
import <module_name>
from <module_name> import <function_name>
import <module_name> as m
from <module_name> import <function_name> as m_function
from <module_name> import *
```

ITERATORS

In this cheatsheet `'<collection>'` can also mean an iterator.

```
<iter> = iter(<collection>)
<iter> = iter(<function>, to_exclusive)  # Sequence of return values until
'to_exclusive'.
<el>   = next(<iter> [, default])    # Raises StopIteration or returns 'default'
on end.
```

GENERATORS

Convenient way to implement the iterator protocol.

```
def count(start, step):
    while True:
        yield start
        start += step
```

```
>>> counter = count(10, 2)
>>> next(counter), next(counter), next(counter)
(10, 12, 14)
```

DECORATORS

A decorator takes a function, adds some functionality and returns it.

```
@decorator_name
def function_that_gets_passed_to_decorator():
    ...
```

DEBUGGER EXAMPLE

Decorator that prints function's name every time it gets called.

```
from functools import wraps

def debug(func):
    @wraps(func)
    def out(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return out

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

- **Wraps is a helper decorator that copies metadata of function add() to function out().**
- **Without it `'add.__name__'` would return `'out'`.**

CLASS

User defined objects are created using the class keyword.

```
class <name>:
    age = 80 # Class Object Attribute
    def __init__(self, a):
        self.a = a # Object Attribute

    @classmethod
    def get_class_name(cls):
        return cls.__name__
```

INHERITANCE

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

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

MULTIPLE INHERITANCE

```
class A: pass
class B: pass
class C(A, B): pass
```

MRO determines the order in which parent classes are traversed when searching for a method:

```
>>> C.mro()
[<class 'C'>, <class 'A'>, <class 'B'>, <class 'object'>]
```

EXCEPTIONS

```
try:
  5/0
except ZeroDivisionError:
  print("No division by zero!")
```

```
while True:
  try:
    x = int(input('Enter your age: '))
  except ValueError:
    print('Oops!  That was no valid number.  Try again...')
  else: # code that depends on the try block running successfully should be
placed in the else block.
    print('Carry on!')
    break
```

RAISING EXCEPTION

```
raise ValueError('some error message')
```

FINALLY

```
try:
  raise KeyboardInterrupt
except:
  print('oops')
finally:
  print('All done!')
```

COMMAND LINE ARGUMENTS

```
import sys
script_name = sys.argv[0]
arguments   = sys.argv[1:]
```

FILE IO

Opens a file and returns a corresponding file object.

```
<file> = open('<path>', mode='r', encoding=None)
```

Modes

- **'r' - Read (default).**
- **'w' - Write (truncate).**
- **'x' - Write or fail if the file already exists.**
- **'a' - Append.**
- **'w+' - Read and write (truncate).**
- **'r+' - Read and write from the start.**
- **'a+' - Read and write from the end.**
- **'t' - Text mode (default).**
- **'b' - Binary mode.**

File

```
<file>.seek(0)                      # Moves to the start of the file.
```

```
<str/bytes> = <file>.readline()     # Returns a line.
<list>      = <file>.readlines()    # Returns a list of lines.
```

```
<file>.write(<str/bytes>)           # Writes a string or bytes object.
<file>.writelines(<list>)           # Writes a list of strings or bytes objects.
```

- **Methods do not add or strip trailing newlines.**

Read Text From File

```
def read_file(filename):
    with open(filename, encoding='utf-8') as file:
        return file.readlines() # or read()

for line in read_file(filename):
  print(line)
```

Write Text To File

```
def write_to_file(filename, text):
    with open(filename, 'w', encoding='utf-8') as file:
        file.write(text)
```

Append Text To File

```
def append_to_file(filename, text):
    with open(filename, 'a', encoding='utf-8') as file:
        file.write(text)
```

USEFUL LIBRARIES

CSV

```
import csv
```

Read Rows From CSV File

```
def read_csv_file(filename):
    with open(filename, encoding='utf-8') as file:
        return csv.reader(file, delimiter=';')
```

Write Rows To CSV File

```
def write_to_csv_file(filename, rows):
    with open(filename, 'w', encoding='utf-8') as file:
        writer = csv.writer(file, delimiter=';')
        writer.writerows(rows)
```

JSON

```
import json
<str>    = json.dumps(<object>, ensure_ascii=True, indent=None)
<object> = json.loads(<str>)
```

Read Object From JSON File

```
def read_json_file(filename):
    with open(filename, encoding='utf-8') as file:
        return json.load(file)
```

Write Object To JSON File

```
def write_to_json_file(filename, an_object):
    with open(filename, 'w', encoding='utf-8') as file:
        json.dump(an_object, file, ensure_ascii=False, indent=2)
```

Pickle

```
import pickle
<bytes>  = pickle.dumps(<object>)
<object> = pickle.loads(<bytes>)
```

Read Object From File

```
def read_pickle_file(filename):
    with open(filename, 'rb') as file:
        return pickle.load(file)
```

Write Object To File

```
def write_to_pickle_file(filename, an_object):
    with open(filename, 'wb') as file:
        pickle.dump(an_object, file)
```

Profile

Basic

```
from time import time
start_time = time()  # Seconds since
...
duration = time() - start_time
```

Math

```
from math import e, pi
from math import cos, acos, sin, asin, tan, atan, degrees, radians
from math import log, log10, log2
from math import inf, nan, isinf, isnan
```

Statistics

```
from statistics import mean, median, variance, pvariance, pstdev
```

Random

```
from random import random, randint, choice, shuffle
random() # random float between 0 and 1
randint(0, 100) # random integer between 0 and 100
random_el = choice([1,2,3,4]) # select a random element from list
shuffle([1,2,3,4]) # shuffles a list
```

Datetime

- **Module 'datetime' provides 'date' `<D>`, 'time' `<T>`, 'datetime' `<DT>` and 'timedelta' `<TD>` classes. All are immutable and hashable.**
- **Time and datetime can be 'aware' `<a>`, meaning they have defined timezone, or 'naive' `<n>`, meaning they don't.**
- **If object is naive it is presumed to be in system's timezone.**

```
from datetime import date, time, datetime, timedelta
from dateutil.tz import UTC, tzlocal, gets
```

Constructors

```
<D>  = date(year, month, day)
<T>  = time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, fold=0)
<DT> = datetime(year, month, day, hour=0, minute=0, second=0, ...)
<TD> = timedelta(days=0, seconds=0, microseconds=0, milliseconds=0,
                 minutes=0, hours=0, weeks=0)
```

- **Use `'<D/DT>.weekday()'` to get the day of the week (Mon == 0).**
- **`'fold=1'` means second pass in case of time jumping back from one hour.**

Now

```
<D/DTn>  = D/DT.today()                # Current local date or naive datetime.
<DTn>    = DT.utcnow()                 # Naive datetime from current UTC time.
<DTa>    = DT.now(<tz>)                # Aware datetime from current tz time.
```

Timezone

```
<tz>      = UTC                          # UTC timezone.
<tz>      = tzlocal()                    # Local timezone.
<tz>      = gettz('<Cont.>/<City>'       # Timezone from 'Continent/City_Name' str.
```

```
<DTa>     = <DT>.astimezone(<tz>)        # Datetime, converted to passed timezone.
<Ta/DTa> = <T/DT>.replace(tzinfo=<tz>)  # Unconverted object with new timezone.
```

Regex

```
import re
<str>   = re.sub(<regex>, new, text, count=0)  # Substitutes all occurrences.
<list>  = re.findall(<regex>, text)            # Returns all occurrences.
<list>  = re.split(<regex>, text, maxsplit=0)  # Use brackets in regex to keep
the matches.
<Match> = re.search(<regex>, text)             # Searches for first occurrence
of pattern.
<Match> = re.match(<regex>, text)              # Searches only at the beginning
of the text.
```

Match Object

```
<str>   = <Match>.group()   # Whole match.
<str>   = <Match>.group(1)  # Part in first bracket.
<tuple> = <Match>.groups()  # All bracketed parts.
<int>   = <Match>.start()   # Start index of a match.
<int>   = <Match>.end()     # Exclusive end index of a match.
```

Special Sequences

Expressions below hold true for strings that contain only ASCII characters. Use capital letters for negation.

```
'\d' == '[0-9]'          # Digit
'\s' == '[ \t\n\r\f\v]'  # Whitespace
'\w' == '[a-zA-Z0-9_]'   # Alphanumeric
```

CREDITS

Inspired by: https://github.com/gto76/python-cheatsheet