Python Cheat Sheet 💻🐍

We created this Python 3 Cheat Sheet initially for students of Complete Python Developer: Zero to Mastery but we're now sharing it with any Python beginners to help them learn and remember common Python syntax and with intermediate and advanced Python developers as a handy reference.

If you've stumbled across this page and are just starting to learn Python, congrats! Python has been around for quite a while but is having a resurgence and has become one of the most popular coding languages in fields like data science, machine learning and web development.

However, if you're stuck in an endless cycle of YouTube tutorials and want to start building real world projects, become a professional Python 3 developer, have fun and actually get hired, then come join the Zero To Mastery Academy and learn Python alongside thousands of students that are you in your exact shoes.

Otherwise, please enjoy this guide and if you'd like to submit any corrections or suggestions, feel free to email us at support@zerotomastery.io

Contents

- Numbers
- Strings
- Boolean
- Lists
- Dictionaries
- Tuples
- Sets
- None

Python Basics:

- Comparison Operators
- Logical Operators
- Loops
- Range
- Enumerate
- Counter
- Named Tuple
- OrderedDict

Functions:

- Functions
- Lambda
- Comprehensions
- Map,Filter,Reduce
- Ternary
- Any,All
- Closures
- Scope

Advanced Python:

- Modules
- Iterators
- Generators
- Decorators
- Class
- Exceptions

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 a
10 % 3  # 1 --> modulo operator - return the reminder. Goo
```

```
# 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 : s

'Hi there ' + 'Timmy' # 'Hi there Timmy' --> This is calle
'*'*10 # **********
```

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

# Basic Methods
'  I am alone '.strip()              # 'I am alone' --> S
'On an island'.strip('d')            # 'On an islan' -->
'but life is good!'.split()          # ['but', 'life', 'i
'Help me'.replace('me', 'you')       # 'Help you' --> Rep
'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
'oh hi there'.count('e')             # 2

```

```
# String Formatting
name1 = 'Andrei'
name2 = 'Sunny'
print(f'Hello there {name1} and {name2}')       # Hello th
print('Hello there {} and {}'.format(name1, name2))# Hello
print('Hello there %s and %s' %(name1, name2))  # Hello th
```

```
#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
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 f
```

Lists

Unlike strings, lists are mutable sequences in python

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

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 : s
```

```
# Add to List
my_list * 2                # [1, 2, '3', True, 1, 2, '3',
my_list + [100]            # [1, 2, '3', True, 100] --> do
my_list.append(100)        # None --> Mutates original lis
my_list.extend([100, 200]) # None --> Mutates original lis
my_list.insert(2, '!!!')   # None -->  [1, 2, '!!!', '3',
```

```
' '.join(['Hello','There'])# 'Hello There' --> Joins eleme
```

```
# 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 in
[1,2,3].pop(1)   # 2 --> mutates original list
[1,2,3].remove(2)# None --> [1,3] Removes first occurrence
[1,2,3].clear()  # None --> mutates original list and remo
del [1,2,3][0] #
```

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

```
# 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

# 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(l
```

```
# Combine columns with zip and *:
[x for x in zip(*mx)] # [(1, 3), (2, 4)]
```

```
# List Comprehensions
# new_list[<action> for <item> in <iterator> if <some cond
a = [i for i in 'hello']                  # ['h', 'e', 'l'
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')
sum_of_elements = sum([1,2,3,4,5])
element_sum = [sum(pair) for pair in zip([1,2,3],[4,5,6])]
sorted_by_second = sorted(['hi','you','man'], key=lambda e
sorted_by_key = sorted([
                        {'name': 'Bina', 'age': 30},
                        {'name':'Andy', 'age': 18},
                        {'name': 'Zoey', 'age': 55}],
                        key=lambda el: (el['name']))# [{'na
```

```
# 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_pow
my_dict['name']                       # Andrei Neagoie
len(my_dict)                          # 3
list(my_dict.keys())                  # ['name', 'age', 'ma
list(my_dict.values())                # ['Andrei Neagoie',
list(my_dict.items())                 # [('name', 'Andrei N
my_dict['favourite_snack'] = 'Grapes'# {'name': 'Andrei Ne
my_dict.get('age')                    # 30 --> Returns None
my_dict.get('ages', 0 )               # 0 --> Returns defau

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

```
my_dict.update({'cool': True})
{**my_dict, **{'cool': True} }
new_dict = dict([['name','Andrei'],['age',32],['magic_powe
new_dict = dict(zip(['name','age','magic_power'],['Andrei'
new_dict = my_dict.pop('favourite_snack')
```

```
# Dictionary Comprehension
{key: value for key, value in new_dict.items() if key == '
```

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(
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 eleme
my_set.discard(100)     # {1} --> Doesn't raise an error i
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 T

```

```
# Frozenset
# hashable --> it can be used as a key in a dictionary or
<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
<                   # left operand is less than right ope
```

```
>=                  # left operand is greater than or equ
<=                  # left operand is less than or equal
<element> is <element> # check if two operands refer to sa
```

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, '
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 re
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,
```

* 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

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

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
factorial = reduce(lambda x, y: x*y, range(1, n+1))
print(factorial) #6
```

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

Comprehensions

```python
<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, .
```

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

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

Ternary Condition

```python
# <expression_if_true> if <condition> else <expression_if_

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

Map Filter Reduce

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

Any All

```
any([False, True, False])# True if at least one item in co
all([True,1,3,True])     # True if all items in collection
```

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 im
    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
<el>   = next(<iter> [, default])           # Raises StopI
```

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

```python 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

```
<h2 id="multiple-inheritance">Multiple Inheritance</h2>
```python
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 succe
    print('Carry on!')
    break
```

Raising Exception

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

Finally

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

```
<h2 id="command-line-arguments">Command Line Arguments</h2

```python
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 o
```

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

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

- **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=
<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, ind
```

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,
from math import log, log10, log2
from math import inf, nan, isinf, isnan
```

Statistics

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

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 fr
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, gettz
```

Constructors

```
<D>  = date(year, month, day)
<T>  = time(hour=0, minute=0, second=0, microsecond=0, tzi
<DT> = datetime(year, month, day, hour=0, minute=0, second
```

```
<TD> = timedelta(days=0, seconds=0, microseconds=0, millis
                 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 for one hour.**

Now

```
<D/DTn>  = D/DT.today()                # Current loca
```


Timezone

```
<tz>     = UTC                         # UTC timezone
<tz>     = tzlocal()                   # Local timezo
<tz>     = gettz('<Cont.>/<City>')     # Timezone fro
```

```
<DTa>    = <DT>.astimezone(<tz>)       # Datetime, co
<Ta/DTa> = <T/DT>.replace(tzinfo=<tz>) # Unconverted
```

Regex

```
import re
<str>   = re.sub(<regex>, new, text, count=0)  # Substitut
<list>  = re.findall(<regex>, text)            # Returns a
<list>  = re.split(<regex>, text, maxsplit=0)  # Use brack
<Match> = re.search(<regex>, text)             # Searches
<Match> = re.match(<regex>, text)              # Searches
```

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 mat
```

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: this repo

Python Notes/Cheat Sheet

Comments
from the hash symbol to the end of a line

Code blocks
Delineated by colons and indented code; and not the curly brackets of C, C++ and Java.

```
def is_fish_as_string(argument):
    if argument:
        return 'fish'
    else:
        return 'not fish'
```

Note: Four spaces per indentation level is the Python standard. Never use tabs: mixing tabs and spaces produces hard-to-find errors. Set your editor to convert tabs to spaces.

Line breaks
Typically, a statement must be on one line. Bracketed code - (), [] or {} - can be split across lines; or (if you must) use a backslash \ at the end of a line to continue a statement on to the next line (but this can result in hard to debug code).

Naming conventions

Style	Use
StudlyCase	Class names
joined_lower	Identifiers, functions; and class methods, attributes
_joined_lower	Internal class attributes
__joined_lower	Private class attributes # this use not recommended
joined_lower **ALL_CAPS**	Constants

Basic object types (not a complete list)

Type	Examples	
None	None	# singleton null object
Boolean	True, False	
integer	-1, 0, 1, sys.maxint	
long	1L, 9787L	# arbitrary length ints
float	3.14159265 inf, float('inf') -inf nan, float('nan')	 # infinity # neg infinity # not a number
complex	2+3j	# note use of j
string	'I am a string', "me too" '''multi-line string''', """+1""" r'raw string', b'ASCII string' u'unicode string'	
tuple	empty = () (1, True, 'dog')	# empty tuple # immutable list
list	empty = [] [1, True, 'dog']	# empty list # mutable list
set	empty = set() set(1, True, 'a')	# the empty set # mutable
dictionary	empty = {} {'a': 'dog', 7: 'seven', True: 1}	# mutable object
file	f = open('filename', 'rb')	

Note: Python has four numeric types (integer, float, long and complex) and several sequence types including strings, lists, tuples, bytearrays, buffers, and range objects.

Operators

Operator	Functionality
+	Addition (also string, tuple, list, and other sequence concatenation)
-	Subtraction (also set difference)
*	Multiplication (also string, tuple, list replication)
/	Division
%	Modulus (also a string format function, but this use deprecated)
//	Integer division rounded towards minus infinity
**	Exponentiation
=, -=, +=, /=, *=, %=, //=, **=	Assignment operators
==, !=, <, <=, >=, >	Boolean comparisons
and, or, not	Boolean operators
in, not in	Containment test operators
is, is not	Object identity operators
|, ^, &, ~	Bitwise: or, xor, and, compliment
<<, >>	Left and right bit shift
;	Inline statement separator # inline statements discouraged

Hint: float('inf') always tests as larger than any number, including integers.

Modules
Modules open up a world of Python extensions that can be imported and used. Access to the functions, variables and classes of a module depend on how the module was imported.

Import method	Access/Use syntax
import math	math.cos(math.pi/3)
import math as m **# import using an alias**	m.cos(m.pi/3)
from math import cos, pi **# only import specifics**	cos(pi/3)
from math import * **# BADish global import**	log(e)

Global imports make for unreadable code!!!

Oft used modules

Module	Purpose
datetime **time**	Date and time functions
math	Core math functions and the constants pi and e
pickle	Serialise objects to a file
os **os.path**	Operating system interfaces
re	A library of Perl-like regular expression operations
string	Useful constants and classes
sys	System parameters and functions
numpy	Numerical python library
pandas	R DataFrames for Python
matplotlib	Plotting/charting for Python

If - flow control

```
if condition:   # for example: if x < 5:
    statements
elif condition: # optional – can be multiple
    statements
else:           # optional
    statements
```

For - flow control

```
for x in iterable:
    statements
else:           # optional completion code
    statements
```

While - flow control

```
while condition:
    statements
else:           # optional completion code
    statements
```

Ternary statement

id = *expression* if *condition* else *expression*

```
x = y if a > b else z - 5
```

Some useful adjuncts:

- pass - a statement that does nothing
- continue - moves to the next loop iteration
- break - to exit for and while loop

Trap: break skips the else completion code

Exceptions – flow control

```
try:
    statements
except (tuple_of_errors): # can be multiple
    statements
else:           # optional no exceptions
    statements
finally:        # optional all
    statements
```

Common exceptions (not a complete list)

Exception	Why it happens
AsserionError	Assert statement failed
AttributeError	Class attribute assignment or reference failed
IOError	Failed I/O operation
ImportError	Failed module import
IndexError	Subscript out of range
KeyError	Dictionary key not found
MemoryError	Ran out of memory
NameError	Name not found
TypeError	Value of the wrong type
ValueError	Right type but wrong value

Raising errors

Errors are raised using the raise statement

```
raise ValueError(value)
```

Creating new errors

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

Objects and variables (AKA identifiers)

- Everything is an object in Python (in the sense that it can be assigned to a variable or passed as an argument to a function)
- Most Python objects have methods and attributes. For example, all functions have the built-in attribute __doc__, which returns the doc string defined in the function's source code.
- All variables are effectively "pointers", not "locations". They are references to objects; and often called identifiers.
- Objects are strongly typed, not identifiers
- Some objects are immutable (int, float, string, tuple, frozenset). But most are mutable (including: list, set, dictionary, NumPy arrays, etc.)
- You can create our own object types by defining a new class (see below).

Booleans and truthiness

Most Python objects have a notion of "truth".

False	True
None	
0 **int(False) # → 0**	Any number other than 0 int(True) # → 1
"" **# the empty string**	" ", 'fred', 'False' # all other strings
() [] {} set() **# empty containers**	[None], (False), {1, 1} # non-empty containers, including those containing False or None.

You can use bool() to discover the truth status of an object.

```
a = bool(obj)          # the truth of obj
```

It is pythonic to use the truth of objects.

```
if container:          # test not empty
    # do something
while items:           # common looping idiom
    item = items.pop()
    # process item
```

Specify the truth of the classes you write using the __nonzero__() magic method.

Comparisons

Python lets you compare ranges, for example

```
assert(1 <= x <= 100)
```

Tuples

Tuples are immutable lists. They can be searched, indexed and iterated much like lists (see below). List methods that do not change the list also work on tuples.

```
a = ()                          # the empty tuple
a = (1,)  # ← note comma        # one item tuple
a = (1, 2, 3)                   # multi-item tuple
a = ((1, 2), (3, 4))            # nested tuple
a = tuple(['a', 'b'])           # conversion
```

Note: the comma is the tuple constructor, not the parentheses. The parentheses add clarity.

The Python swap variable idiom

```
a, b = b, a    # no need for a temp variable
```

This syntax uses tuples to achieve its magic.

String (immutable, ordered, characters)

```
s = 'string'.upper()          # STRING
s = 'fred'+'was'+'here'       # concatenation
s = ''.join(['fred', 'was', 'here'])       # ditto
s = 'spam' * 3                # replication
s = str(x)                    # conversion
```

String iteration and sub-string searching

```
for character in 'str':       # iteration
    print (ord(character))    # 115 116 114
for index, character in enumerate('str')
    print (index, character)
if 'red' in 'Fred':           # searching
    print ('Fred is red')     # it prints!
```

String methods (not a complete list)

capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill

String constants (not a complete list)

```
from string import *  # global import is not good
print ([digits, hexdigits, ascii_letters,
    ascii_lowercase, ascii_uppercase,
    punctuation])
```

Old school string formatting (using % oper)

```
print ("It %s %d times" % ('occurred', 5))
# prints: 'It occurred 5 times'
```

Code	Meaning
s	String or string conversion
c	Character
d	Signed decimal integer
u	Unsigned decimal integer
H or h	Hex integer (upper or lower case)
f	Floating point
E or e	Exponent (upper or lower case E)
G or g	The shorter of e and f (u/l case)
%	Literal '%'

```
import math
'%s' % math.pi        # '3.14159265359'
'%f' % math.pi        # '3.141593'
'%.2f' % math.pi      # '3.14'
'%.2e' % 3000         # '3.00e+03'
'%03d' % 5            # '005'
```

New string formatting (using format method)

Uses: 'template-string'.format(arguments)
Examples (using similar codes as above):

```
import math
'Hello {}'.format('World')   # 'Hello World'
'{}'.format(math.pi)         # ' 3.14159265359'
'{0:.2f}'.format(math.pi)    # '3.14'
'{0:+.2f}'.format(5)         # '+5.00'
'{:.2e}'.format(3000)        # '3.00e+03'
'{:0>2d}'.format(5)          # '05' (left pad)
'{:x<3d}'.format(5)          # '5xx' (rt. pad)
'{:,}'.format(1000000)       # '1,000,000'
'{:.1%}'.format(0.25)        # '25.0%'
'{0}{1}'.format('a', 'b')    # 'ab'
'{1}{0}'.format('a', 'b')    # 'ba'
'{num:}'.format(num=7)       # '7' (named args)
```

List (mutable, indexed, ordered container)

Indexed from zero to length-1

```
a = []                        # the empty list
a = ['dog', 'cat', 'bird']    # simple list
a = [[1, 2], ['a', 'b']]      # nested lists
a = [1, 2, 3] + [4, 5, 6]     # concatenation
a = [1, 2, 3] * 456           # replication
a = list(x)                   # conversion
```

List comprehensions (can be nested)

Comprehensions: a tight way of creating lists

```
t3 = [x*3 for x in [5, 6, 7]] # [15, 18, 21]
z = [complex(x, y) for x in range(0, 4, 1)
          for y in range(4, 0, -1) if x > y]
# z --> [(2+1j), (3+2j), (3+1j)]
```

Iterating lists

```
L = ['dog', 'cat', 'turtle']
for item in L
    print (item)
for index, item in enumerate(L):
    print (index, item)
```

Searching lists

```
L = ['dog', 'cat', 'turtle']
value = 'cat'
if value in L:
    count = L.count(value)
    first_occurrence = L.index(value)
if value not in L:
    print 'list is missing {}'.format(value)
```

List methods (not a complete list)

Method	What it does
l.append(x)	Add x to end of list
l.extend(other)	Append items from other
l.insert(pos, x)	Insert x at position
del l[pos]	Delete item at pos
l.remove(x)	Remove first occurrence of x; An error if no x
l.pop([pos])	Remove last item from list (or item from pos); An error if empty list
l.index(x)	Get index of first occurrence of x; An error if x not found
l.count(x)	Count the number of times x is found in the list
l.sort()	In place list sort
l.reverse(x)	In place list reversal

List slicing

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8] # play data
x[2]     # 3rd element - reference not slice
x[1:3]   # 2nd to 3rd element (1, 2)
x[:3]    # the first three elements (0, 1, 2)
x[-3:]   # last three elements
x[:-3]   # all but the last three elements
x[:]     # every element of x – copies x
x[1:-1]  # all but first and last element
x[::3]   # (0, 3, 6, 9, …) 1st then every 3rd
x[1:5:2] # (1,3) start 1, stop >= 5, by every 2nd
```

Note: All Python sequence types support the above index slicing (strings, lists, tuples, bytearrays, buffers, and xrange objects)

Set (unique, unordered container)

A Python set is an unordered, mutable collection of unique hashable objects.

```
a = set()                    # empty set
a = {'red', 'white', 'blue'} # simple set
a = set(x)                   # convert list
```

Trap: {} creates empty dict, not an empty set

Set comprehensions

```
# a set of selected letters...
s = {e for e in 'ABCHJADC' if e not in 'AB'}
# --> {'H', 'C', 'J', 'D'}
# a set of tuples ...
s = {(x,y) for x in range(-1,2)
           for y in range (-1,2)}
```

Trap: set contents need to be immutable to be hashable. So you can have a set of tuples, but not a set of lists.

Iterating a set

```
for item in set:
    print (item)
```

Searching a set

```
if item in set:
    print (item)
if item not in set:
    print ('{} is missing'.format(item))
```

Set methods (not a complete list)

Method	What it does
len(s)	Number of items in set
s.add(item)	Add item to set
s.remove(item)	Remove item from set. Raise KeyError if item not found.
s.discard(item)	Remove item from set if present.
s.pop()	Remove and return an arbitrary item. Raise KeyError on empty set.
s.clear()	Remove all items from set
item in s	True or False
item not in s	True or False
iter(s)	An iterator over the items in the set (arbitrary order)
s.copy()	Get shallow copy of set
s.isdisjoint(o)	True if s has not items in common with other set o
s.issubset(o)	Same as set <= other
s.issuperset(o)	Same as set >= other
s.union(o[, ...])	Return new union set
s.intersection(o)	Return new intersection
s.difference(o)	Get net set of items in s but not others (Same as set – other)

Frozenset

Similar to a Python set above, but immutable (and therefore hashable).

```
f = frozenset(s)             # convert set
f = frozenset(o)             # convert other
```

Dictionary (indexed, unordered map-container)

A mutable hash map of unique key=value pairs.

```
a = {}                       # empty dictionary
a = {1: 1, 2: 4, 3: 9}       # simple dict
a = dict(x)                  # convert paired data
# next example – create from a list
l = ['alpha', 'beta', 'gamma', 'delta']
a = dict(zip(range(len(l)), l))
# Example using string & generator expression
s = 'a=apple,b=bird,c=cat,d=dog,e=egg'
a = dict(i.split("=") for i in s.split(","))
# {'a': 'apple', 'c': 'cat', 'b': 'bird',
#    'e': 'egg', 'd': 'dog'}
```

Dictionary comprehensions

Conceptually like list comprehensions; but it constructs a dictionary rather than a list

```
a = { n: n*n for n in range(7) }
# a -> {0:0, 1:1, 2:4, 3:9, 4:16, 5:25,6:36}
odd_sq = { n: n*n for n in range(7) if n%2 }
# odd_sq -> {1: 1, 3: 9, 5: 25}
# next example -> swaps the key:value pairs
a = { val:key for key, val in a.items() }
# next example -> count list occurrences
l = [11,12,13,11,15,19,15,11,20,13,11,11,12,10]
c = { key: l.count(key) for key in set(l) }
```

Iterating a dictionary

```
for key in dictionary.keys():
    print (key)
for key, value in dictionary.items():
    print (key, value)
for value in dictionary.values():
    print(value)
```

Searching a dictionary

```
if key in dictionary:
    print (key)
```

Merging two dictionaries

```
merged = dict_1.copy()
merged.update(dict_2)
```

Dictionary methods (not a complete list)

Method	What it does
len(d)	Number of items in d
d[key]	Get value for key or raise the KeyError exception
d[key] = value	Set key to value
del d[key]	deletion
key in d	True or False
key not in d	True or False
iter(d)	An iterator over the keys
d.clear()	Remove all items from d
d.copy()	Shallow copy of dictionary
d.get(key[, def])	Get value else default
d.items()	Dictionary's (k,v) pairs
d.keys()	Dictionary's keys
d.pop(key[, def])	Get value else default; remove key from dictionary
d.popitem()	Remove and return an arbitrary (k, v) pair
d.setdefault(k[,def]))	If k in dict return its value otherwise set def
d.update(other_d)	Update d with key:val pairs from other
d.values()	The values from dict

Key functions (not a complete list)

Function	What it does
abs(num)	Absolute value of num
all(iterable)	True if all are True
any(iterable)	True if any are True
bytearray(source)	A mutable array of bytes
callable(obj)	True if obj is callable
chr(int)	Character for ASCII int
complex(re[, im])	Create a complex number
divmod(a, b)	Get (quotient, remainder)
enumerate(seq)	Get an enumerate object, with next() method returns an (index, element) tuple
eval(string)	Evaluate an expression
filter(fn, iter)	Construct a list of elements from *iter* for which *fn()* returns True
float(x)	Convert from int/string
getattr(obj, str)	Like obj.str
hasattr(obj, str)	True if obj has attribute
hex(x)	From in to hex string
id(obj)	Return unique (run-time) identifier for an object
int(x)	Convert from float/string
isinstance(o, c)	Eg. isinstance(2.1, float)
len(x)	Number of items in x; x is string, tuple, list, dict
list(iterable)	Make a list
long(x)	Convert a string or number to a long integer
map(fn, iterable)	Apply fn() to every item in iterable; return results in a list
max(a,b) **max(iterable)**	What it says on the tin
min(a,b) **min(iterable)**	Ditto
next(iterator)	Get next item from an iter
open(name[,mode])	Open a file object
ord(c)	Opposite of chr(int)
pow(x, y)	Same as x ** y
print (objects)	What it says on the tin takes end arg (default \n) and sep arg (default ' ')
range(stop) **range(start,stop)** **range(fr,to,step)**	integer list; stops < stop default start=0; default step=1
reduce(fn, iter)	Applies the two argument fn(x, y) cumulatively to the items of iter.
repr(object)	Printable representation of an object
reversed(seq)	Get a reversed iterator
round(n[,digits])	Round to number of digits after the decimal place
setattr(obj,n,v)	Like obj.n = v #name/value
sorted(iterable)	Get new sorted list
str(object)	Get a string for an object
sum(iterable)	Sum list of numbers
type(object)	Get the type of object
xrange()	Like range() but better: returns an iterator
zip(x, y[, z])	Return a list of tuples

Using functions

When called, functions can take positional and named arguments.

For example:

```
result = function(32, aVar, c='see', d={})
```

Arguments are passed by reference (ie. the objects are not copied, just the references).

Writing a simple function

```
def funct(arg1, arg2=None, *args, **kwargs):
    """explain what this function does"""
    statements
    return x     # optional statement
```

Note: functions are first class objects that get instantiated with attributes and they can be referenced by variables.

Avoid named default mutable arguments

Avoid mutable objects as default arguments. Expressions in default arguments are evaluated when the function is defined, not when it's called. Changes to mutable default arguments survive between function calls.

```
def nasty(value=[]):       # <-- mutable arg
    value.append('a')
    return value
print (nasty ()) # --> ['a']
print (nasty ()) # --> ['a', 'a']

def better(val=None):
    val = [] if val is None else val
    value.append('a')
    return value
```

Lambda (inline expression) functions:

```
g = lambda x: x ** 2      # Note: no return
print(g(8))               # prints 64
mul = lambda a, b: a * b  # two arguments
mul(4, 5) == 4 * 5        # --> True
```

Note: only for expressions, not statements.
Lambdas are often used with the Python functions filter(), map() and reduce().

```
# get only those numbers divisible by three
div3 = filter(lambda x: x%3==0,range(1,101))
```

Typically, you can put a lambda function anywhere you put a normal function call.

Closures

Closures are functions that have inner functions with data fixed in the inner function by the lexical scope of the outer. They are useful for avoiding hard constants. Wikipedia has a derivative function for changeable values of dx, using a closure.

```
def derivative(f, dx):
    """Return a function that approximates
       the derivative of f using an interval
       of dx, which should be appropriately
       small.
    """
    def _function(x):
        return (f(x + dx) - f(x)) / dx
    return _function #from derivative(f, dx)

f_dash_x = derivative(lambda x: x*x,0.00001)
f_dash_x(5) # yields approx. 10 (ie. y'=2x)
```

An iterable object
The contents of an iterable object can be selected one at a time. Such objects include the Python sequence types and classes with the magic method __iter__(), which returns an iterator. An iterable object will produce a fresh iterator with each call to iter().

```
iterator = iter(iterable_object)
```

Iterators
Objects with a next() or __next__() method, that:

- returns the next value in the iteration
- updates the internal note of the next value
- raises a StopIteration exception when done

Note: with the loop for x in y: if y is not an iterator; Python calls iter() to get one. With each loop, it calls next() on the iterator until a StopIteration exception.

```
x = iter('XY')  # iterate a string by hand
print (next(x)) # --> X
print (next(x)) # --> Y
print (next(x)) # --> StopIteration exception
```

Generators
Generator functions are resumable functions that work like iterators. They can be more space or time efficient than iterating over a list, (especially a very large list), as they only produce items as they are needed.

```
def fib(max=None):
    """ generator for Fibonacci sequence"""
    a, b = 0, 1
    while max is None or b <= max:
        yield b    # ← yield is like return
        a, b = b, a+b

[i for i in fib(10)] # → [1, 1, 2, 3, 5, 8]
```

Note: a return statement (or getting to the end of the function) ends the iteration.
Trap: a yield statement is not allowed in the try clause of a try/finally construct.

Messaging the generator

```
def resetableCounter(max=None):
    j = 0
    while max is None or j <= max:
        x = yield j # ← x gets the sent arg
        j = j+1 if x is None else x

x = resetableCounter(10)
print x.send(None)      # → 0
print x.send(5)         # → 5
print x.send(None)      # → 6
print x.send(11)        # → StopIteration
```

Trap: must send None on first send() call

Generator expressions
Generator expressions build generators, just like building a list from a comprehension. You can turn a list comprehension into a generator expression simply by replacing the square brackets [] with parentheses ().

```
[i for i in range(10)]      # list comprehension
list(i for i in range(10))  # generated list
```

Classes
Python is an object-oriented language with a multiple inheritance class mechanism that encapsulates program code and data.

Methods and attributes
Most objects have associated functions or “methods” that are called using dot syntax:

```
obj.method(arg)
```

Objects also often have attributes or values that are directly accessed without using getters and setters (most unlike Java or C++)

```
instance = Example_Class()
print (instance.attribute)
```

Simple example

```
import math
class Point:
    # static class variable, point count
    count = 0

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)
        Point.count += 1

    def __str__(self):
        return \
        '(x={}, y={})'.format(self.x, self.y)

    def to_polar(self):
        r = math.sqrt(self.x**2 + self.y**2)
        theta = math.atan2(self.y, self.x)
        return(r, theta)

    # static method – trivial example ...
    def static_eg(n):
        print ('{}'.format(n))
    static_eg = staticmethod(static_eg)

# Instantiate 9 points & get polar coords
for x in range(-1, 2):
    for y in range(-1, 2):
        p = Point(x, y)
        print (p)   # uses __str__() method
        print (p.to_polar())
print (Point.count) # check static variable
Point.static_eg(9)  # check static method
```

The self
Class methods have an extra argument over functions. Usually named 'self'; it is a reference to the instance. It is not used in the method call; and is provided by Python to the method. Self is like 'this' in C++ & Java

Public and private methods and variables
Python does not enforce the public v private data distinction. By convention, variables and methods that begin with an underscore should be treated as private (unless you really know what you are doing). Variables that begin with double underscore are mangled by the compiler (and hence more private).

Inheritance

```
class DerivedClass1(BaseClass):
    statements
class DerivedClass2(module_name.BaseClass):
    statements
```

Multiple inheritance

```
class DerivedClass(Base1, Base2, Base3):
    statements
```

Decorators

Technically, decorators are just functions (or classes), that take a callable object as an argument, and return an analogous object with the decoration. We will skip how to write them, and focus on using a couple of common built in decorators.

Practically, decorators are syntactic sugar for more readable code. The *@wrapper* is used to transform the existing code. For example, the following two method definitions are semantically equivalent.

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Getters and setters

Although class attributes can be directly accessed, the property function creates a property manager.

```
class Example:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx,"Doc txt")
```

Which can be rewritten with decorators as:

```
class Example:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """Doc txt: I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Magic class methods (not a complete list)

Magic methods (which begin and end with double underscore) add functionality to your classes consistent with the broader language.

Magic method	What it does
__init__(self,[...])	Constructor
__del__(self)	Destructor pre-garbage collection
__str__(self)	Human readable string for class contents. Called by str(self)
__repr__(self)	Machine readable unambiguous Python string expression for class contents. Called by repr(self) Note: str(self) will call __repr__ if __str__ is not defined.
__eq__(self, other)	Behaviour for ==
__ne__(self, other)	Behaviour for !=
__lt__(self, other)	Behaviour for <
__gt__(self, other)	Behaviour for >
__le__(self, other)	Behaviour for <=
__ge__(self, other)	Behaviour for >=
__add__(self, other)	Behaviour for +
__sub__(self, other)	Behaviour for -
__mul__(self, other)	Behaviour for *
__div__(self, other)	Behaviour for /
__mod__(self, other)	Behaviour for %
__pow__(self, other)	Behaviour for **
__pos__(self, other)	Behaviour for unary +
__neg__(self, other)	Behaviour for unary -
__hash__(self)	Returns an int when hash() called. Allows class instance to be put in a dictionary
__len__(self)	Length of container
__contains__(self, i)	Behaviour for in and not in operators
__missing__(self, i)	What to do when dict key i is missing
__copy__(self)	Shallow copy constructor
__deepcopy__(self, memodict={})	Deep copy constructor
__iter__(self)	Provide an iterator
__nonzero__(self)	Called by bool(self)
__index__(self)	Called by x[self]
__setattr__(self, name, val)	Called by self.name = val
__getattribute__(self, name)	Called by self.name
__getattr__(self, name)	Called when self.name does not exist
__delattr__(self, name)	Called by del self.name
__getitem__(self, key)	Called by self[key]
__setitem__(self, key, val)	Called by self[key] = val
__delitem__(self, key)	del self[key]

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```

Conditional test with lists

```
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

While loops

A while loop repeats a block of code as long as a certain condition is true.

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")

make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:
    print(user)
```

Printing a message for each item, and a separate message afterwards

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```

The range() function

You can use the range() function to work with a set of numbers efficiently. The range() function starts at 0 by default, and stops one number below the number passed to it. You can use the list() function to efficiently generate a large list of numbers.

Printing the numbers 0 to 1000

```
for number in range(1001):
    print(number)
```

Printing the numbers 1 to 1000

```
for number in range(1, 1001):
    print(number)
```

Making a list of numbers from 1 to a million

```
numbers = list(range(1, 1000001))
```

Simple statistics

There are a number of simple statistics you can run on a list containing numerical data.

Finding the minimum value in a list

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

Finding the maximum value

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
oldest = max(ages)
```

Finding the sum of all values

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

Slicing a list

You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at the beginning of the list, and leave off the last index to slice through the end of the list.

Getting the first three items

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

Getting the middle three items

```
middle_three = finishers[1:4]
```

Getting the last three items

```
last_three = finishers[-3:]
```

Copying a list

To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.

Making a copy of a list

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

List comprehensions

You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.

To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.

Using a loop to generate a list of square numbers

```
squares = []
for x in range(1, 11):
    square = x**2
    squares.append(square)
```

Using a comprehension to generate a list of square numbers

```
squares = [x**2 for x in range(1, 11)]
```

Using a loop to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names.append(name.upper())
```

Using a comprehension to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

Styling your code

Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

Tuples

A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)

Defining a tuple

```
dimensions = (800, 600)
```

Looping through a tuple

```
for dimension in dimensions:
    print(dimension)
```

Overwriting a tuple

```
dimensions = (800, 600)
print(dimensions)

dimensions = (1200, 900)
```

Visualizing your code

When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. pythontutor.com is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on pythontutor.com, and then run your own code.

Build a list and print the items in the list

```
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print("Hello " + dog + "!")
print("I love these dogs!")

print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)

del dogs[0]
dogs.remove('peso')
print(dogs)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

Defining a dictionary

Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.

Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.

You can also use the get() method, which returns None instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.

Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

Getting the value with get()

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)

print(alien_color)
print(alien_points)
```

Adding new key-value pairs

You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.

This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.

Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

Modifying values

You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.

Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

Removing key-value pairs

You can remove any key-value pair you want from a dictionary. To do so use the del keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.

Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.

A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.

Looping through all key-value pairs

```
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Looping through all the keys

```
# Show everyone who's taken the survey.
for name in fav_languages.keys():
    print(name)
```

Looping through all the values

```
# Show all the languages that have been chosen.
for language in fav_languages.values():
    print(language)
```

Looping through all the keys in order

```
# Show each person's favorite language,
#   in order by the person's name.
for name in sorted(fav_languages.keys()):
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

Finding a dictionary's length

```
num_responses = len(fav_languages)
```

Nesting — A list of dictionaries

It's sometimes useful to store a set of dictionaries in a list; this is called nesting.

Storing dictionaries in a list

```
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
    }
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

You can also define a list of dictionaries directly, without using append():

```
# Define a list of users, where each user
#   is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

Nesting — Lists in a dictionary

Storing a list inside a dictionary alows you to associate more than one value with each key.

Storing lists in a dictionary

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.

Storing dictionaries in a dictionary

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },
    }

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

Levels of nesting

Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.

Using an OrderedDict

Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an OrderedDict.

Preserving the order of keys and values

```
from collections import OrderedDict

# Store each person's languages, keeping
#  track of who respoded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
#  were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

Generating a million dictionaries

You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.

A million aliens

```
aliens = []

# Make a million green aliens, worth 5 points
#  each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12

if age < 4:
   price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

Conditional tests with lists (cont.)

Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'

if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```
players = []

if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.

Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7

Use raw_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

A while loop repeats a block of code as long as a condition is True.

Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.

Breaking out of loops

You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.

An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.

Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.

Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

Visualizing functions

Try running some of these examples on pythontutor.com.

Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list

The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling print_models().

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

*Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the * operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.*

*The ** operator allows a parameter to collect an arbitrary number of keyword arguments.*

Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
        'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
#   of information.
user_0 = build_profile('albert', 'einstein',
        location='princeton')
user_1 = build_profile('marie', 'curie',
        location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

Storing a function in a module

File: pizza.py

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module

File: making_pizzas.py
Every function in the module is available in the program file.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module

Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The __init__() method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.

Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model complex situations.

A Battery class

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file

car.py

```
"""Represent gas and electric cars."""

class Car():
    """A simple attempt to model a car."""
    --snip—

class Battery():
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s',
2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
        'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module

(Don't do this, but recognize it when you see it.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```
class ClassName(object):
```

The Car class in Python 2.7

```
class Car(object):
```

Child class __init__() method is different

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
                make, model, year)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.

Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.

A fleet of rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

The else block

The try block should only contain code that may cause an error. Any code that depends on the try block running successfully should be placed in the else block.

Using an else block

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input

Without the except block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the pass statement in an else block allows you to do this.

Using the pass statement in an else block

```
f_names = ['alice.txt', 'siddhartha.txt',
        'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare except block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a try block and you're not sure which exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use Exception instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

The json module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using json.dump() to store data

```
"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using json.load() to read data

```
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions

Take a program you've already written that prompts for user input, and add some error-handling code to the program.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Testing Your Code

Why test your code?

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know that you haven't broken existing behavior.

A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is a collection of unit tests which verify your code's behavior in a wide variety of situations.

Testing a function: A passing test

Python's unittest module provides tools for testing your code. To try it out, we'll create a function that returns a full name. We'll use the function in a regular program, and then build a test case for the function.

A function to test

Save this as full_names.py

```
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function

Save this as names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Testing a function (cont.)

Building a testcase with one unit test

To build a test case, make a class that inherits from `unittest.TestCase` *and write methods that begin with* `test_`*. Save this as test_full_names.py*

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

unittest.main()
```

Running the test

Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

Testing a function: A failing test

Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.

Modifying the function

We'll modify `get_full_name()` *so it handles middle names, but we'll do it in a way that breaks existing behavior.*

```
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                     middle, last)
    return full_name.title()
```

Using the function

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test

When you change your code, it's important to run your existing tests. This will tell you whether the changes you made affected existing behavior.

```
E
======================================================
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
------------------------------------------------------
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
  positional argument: 'last'

------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Fixing the code

When a test fails, the code needs to be modified until the test passes again. (Don't make the mistake of rewriting your tests to fit your new code.) Here we can make the middle name optional.

```
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                                         middle, last)
    else:
        full_name = "{0} {1}".format(first,
                                     last)
    return full_name.title()
```

Running the test

Now the test should pass again, which means our original functionality is still intact.

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

Adding new tests

You can add as many unit tests to a test case as you need. To write a new test, add a new method to your test case class.

Testing middle names

We've shown that `get_full_name()` works for first and last names. Let's test that it works for middle names as well.

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

    def test_middle(self):
        """Test names like David Lee Roth."""
        full_name = get_full_name('david',
                'roth', 'lee')
        self.assertEqual(full_name,
                'David Lee Roth')

unittest.main()
```

Running the tests

The two dots represent two passing tests.

```
..
--------------------------------------
Ran 2 tests in 0.000s

OK
```

A variety of assert methods

Python provides a number of assert methods you can use to test your code.

Verify that a==b, or a != b

```
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that x is True, or x is False

```
assertTrue(x)
assertFalse(x)
```

Verify an item is in a list, or not in a list

```
assertIn(item, list)
assertNotIn(item, list)
```

Testing a class

Testing a class is similar to testing a function, since you'll mostly be testing your methods.

A class to test

Save as accountant.py

```
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as test_accountant.py.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

When is it okay to modify tests?

In general you shouldn't modify a test once it's written. When a test fails it usually means new code you've written has broken existing functionality, and you need to modify the new code until all existing tests pass.

If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.

The setUp() method

When testing a class, you usually have to make an instance of the class. The `setUp()` method is run before every test. Any instances you make in `setUp()` are available in every test you write.

Using setUp() to support multiple tests

The instance `self.acc` can be used in each new test.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def setUp(self):
        self.acc = Accountant()

    def test_initial_balance(self):
        # Default balance should be 0.
        self.assertEqual(self.acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)

unittest.main()
```

Running the tests

```
...
--------------------------------------
Ran 3 tests in 0.001s

OK
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Pygame

What is Pygame?

Pygame is a framework for making games using Python. Making games is fun, and it's a great way to expand your programming skills and knowledge. Pygame takes care of many of the lower-level tasks in building games, which lets you focus on the aspects of your game that make it interesting.

Installing Pygame

Pygame runs on all systems, but setup is slightly different on each OS. The instructions here assume you're using Python 3, and provide a minimal installation of Pygame. If these instructions don't work for your system, see the more detailed notes at http://ehmatthes.github.io/pcc/.

Pygame on Linux

```
$ sudo apt-get install python3-dev mercurial
    libsdl-image1.2-dev libsdl2-dev
    libsdl-ttf2.0-dev
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on OS X

This assumes you've used Homebrew to install Python 3.

```
$ brew install hg sdl sdl_image sdl_ttf
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on Windows

Find an installer at https://bitbucket.org/pygame/pygame/downloads/ or http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame that matches your version of Python. Run the installer file if it's a .exe or .msi file. If it's a .whl file, use pip to install Pygame:

```
> python -m pip install --user
    pygame-1.9.2a0-cp35-none-win32.whl
```

Testing your installation

To test your installation, open a terminal session and try to import Pygame. If you don't get any error messages, your installation was successful.

```
$ python
>>> import pygame
>>>
```

Starting a game

The following code sets up an empty game window, and starts an event loop and a loop that continually refreshes the screen.

An empty game window

```
import sys
import pygame as pg

def run_game():
    # Initialize and set up screen.
    pg.init()
    screen = pg.display.set_mode((1200, 800))
    pg.display.set_caption("Alien Invasion")

    # Start main loop.
    while True:
        # Start event loop.
        for event in pg.event.get():
            if event.type == pg.QUIT:
                sys.exit()

        # Refresh screen.
        pg.display.flip()

run_game()
```

Setting a custom window size

The display.set_mode() function accepts a tuple that defines the screen size.

```
screen_dim = (1200, 800)
screen = pg.display.set_mode(screen_dim)
```

Setting a custom background color

Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255.

```
bg_color = (230, 230, 230)
screen.fill(bg_color)
```

Pygame rect objects

Many objects in a game can be treated as simple rectangles, rather than their actual shape. This simplifies code without noticeably affecting game play. Pygame has a rect object that makes it easy to work with game objects.

Getting the screen rect object

We already have a screen object; we can easily access the rect object associated with the screen.

```
screen_rect = screen.get_rect()
```

Finding the center of the screen

Rect objects have a center attribute which stores the center point.

```
screen_center = screen_rect.center
```

Pygame rect objects (cont.)

Useful rect attributes

Once you have a rect object, there are a number of attributes that are useful when positioning objects and detecting relative positions of objects. (You can find more attributes in the Pygame documentation.)

```
# Individual x and y values:
screen_rect.left, screen_rect.right
screen_rect.top, screen_rect.bottom
screen_rect.centerx, screen_rect.centery
screen_rect.width, screen_rect.height

# Tuples
screen_rect.center
screen_rect.size
```

Creating a rect object

You can create a rect object from scratch. For example a small rect object that's filled in can represent a bullet in a game. The Rect() class takes the coordinates of the upper left corner, and the width and height of the rect. The draw.rect() function takes a screen object, a color, and a rect. This function fills the given rect with the given color.

```
bullet_rect = pg.Rect(100, 100, 3, 15)
color = (100, 100, 100)
pg.draw.rect(screen, color, bullet_rect)
```

Working with images

Many objects in a game are images that are moved around the screen. It's easiest to use bitmap (.bmp) image files, but you can also configure your system to work with jpg, png, and gif files as well.

Loading an image

```
ship = pg.image.load('images/ship.bmp')
```

Getting the rect object from an image

```
ship_rect = ship.get_rect()
```

Positioning an image

With rects, it's easy to position an image wherever you want on the screen, or in relation to another object. The following code positions a ship object at the bottom center of the screen.

```
ship_rect.midbottom = screen_rect.midbottom
```

Working with images (cont.)

Drawing an image to the screen

Once an image is loaded and positioned, you can draw it to the screen with the blit() method. The blit() method acts on the screen object, and takes the image object and image rect as arguments.

```
# Draw ship to screen.
screen.blit(ship, ship_rect)
```

The blitme() method

Game objects such as ships are often written as classes. Then a blitme() method is usually defined, which draws the object to the screen.

```
def blitme(self):
    """Draw ship at current location."""
    self.screen.blit(self.image, self.rect)
```

Responding to keyboard input

Pygame watches for events such as key presses and mouse actions. You can detect any event you care about in the event loop, and respond with any action that's appropriate for your game.

Responding to key presses

Pygame's main event loop registers a KEYDOWN event any time a key is pressed. When this happens, you can check for specific keys.

```
for event in pg.event.get():
    if event.type == pg.KEYDOWN:
        if event.key == pg.K_RIGHT:
            ship_rect.x += 1
        elif event.key == pg.K_LEFT:
            ship_rect.x -= 1
        elif event.key == pg.K_SPACE:
            ship.fire_bullet()
        elif event.key == pg.K_q:
            sys.exit()
```

Responding to released keys

When the user releases a key, a KEYUP event is triggered.

```
if event.type == pg.KEYUP:
    if event.key == pg.K_RIGHT:
        ship.moving_right = False
```

Pygame documentation

The Pygame documentation is really helpful when building your own games. The home page for the Pygame project is at http://pygame.org/, and the home page for the documentation is at http://pygame.org/docs/.

The most useful part of the documentation are the pages about specific parts of Pygame, such as the Rect() class and the sprite module. You can find a list of these elements at the top of the help pages.

Responding to mouse events

Pygame's event loop registers an event any time the mouse moves, or a mouse button is pressed or released.

Responding to the mouse button

```
for event in pg.event.get():
    if event.type == pg.MOUSEBUTTONDOWN:
        ship.fire_bullet()
```

Finding the mouse position

The mouse position is returned as a tuple.

```
mouse_pos = pg.mouse.get_pos()
```

Clicking a button

You might want to know if the cursor is over an object such as a button. The rect.collidepoint() method returns true when a point is inside a rect object.

```
if button_rect.collidepoint(mouse_pos):
    start_game()
```

Hiding the mouse

```
pg.mouse.set_visible(False)
```

Pygame groups

Pygame has a Group class which makes working with a group of similar objects easier. A group is like a list, with some extra functionality that's helpful when building games.

Making and filling a group

An object that will be placed in a group must inherit from Sprite.

```
from pygame.sprite import Sprite, Group

def Bullet(Sprite):
    ...
    def draw_bullet(self):
        ...
    def update(self):
        ...

bullets = Group()

new_bullet = Bullet()
bullets.add(new_bullet)
```

Looping through the items in a group

The sprites() method returns all the members of a group.

```
for bullet in bullets.sprites():
    bullet.draw_bullet()
```

Calling update() on a group

Calling update() on a group automatically calls update() on each member of the group.

```
bullets.update()
```

Pygame groups (cont.)

Removing an item from a group

It's important to delete elements that will never appear again in the game, so you don't waste memory and resources.

```
bullets.remove(bullet)
```

Detecting collisions

You can detect when a single object collides with any member of a group. You can also detect when any member of one group collides with a member of another group.

Collisions between a single object and a group

The spritecollideany() function takes an object and a group, and returns True if the object overlaps with any member of the group.

```
if pg.sprite.spritecollideany(ship, aliens):
    ships_left -= 1
```

Collisions between two groups

The sprite.groupcollide() function takes two groups, and two booleans. The function returns a dictionary containing information about the members that have collided. The booleans tell Pygame whether to delete the members of either group that have collided.

```
collisions = pg.sprite.groupcollide(
        bullets, aliens, True, True)

score += len(collisions) * alien_point_value
```

Rendering text

You can use text for a variety of purposes in a game. For example you can share information with players, and you can display a score.

Displaying a message

The following code defines a message, then a color for the text and the background color for the message. A font is defined using the default system font, with a font size of 48. The font.render() function is used to create an image of the message, and we get the rect object associated with the image. We then center the image on the screen and display it.

```
msg = "Play again?"
msg_color = (100, 100, 100)
bg_color = (230, 230, 230)

f = pg.font.SysFont(None, 48)
msg_image = f.render(msg, True, msg_color,
        bg_color)
msg_image_rect = msg_image.get_rect()
msg_image_rect.center = screen_rect.center
screen.blit(msg_image, msg_image_rect)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — matplotlib

What is matplotlib?

Data visualization involves exploring data through visual representations. The matplotlib package helps you make visually appealing representations of the data you're working with. matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.

Installing matplotlib

matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here don't work for you, see the more detailed instructions at http://ehmatthes.github.io/pcc/. You should also consider installing the Anaconda distrubution of Python from https://continuum.io/downloads/, which includes matplotlib.

matplotlib on Linux

```
$ sudo apt-get install python3-matplotlib
```

matplotlib on OS X

Start a terminal session and enter `import matplotlib` *to see if it's already installed on your system. If not, try this command:*

```
$ pip install --user matplotlib
```

matplotlib on Windows

You first need to install Visual Studio, which you can do from https://dev.windows.com/. The Community edition is free. Then go to https://pypi.python.org/pypi/matplotlib/ or http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib and download an appropriate installer file.

Line graphs and scatter plots

Making a line graph

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
plt.plot(x_values, squares)
plt.show()
```

Line graphs and scatter plots (cont.)

Making a scatter plot

The scatter() function takes a list of x values and a list of y values, and a variety of optional arguments. The `s=10` *argument controls the size of each point.*

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.

Adding titles and labels, and scaling axes

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, s=10)

plt.title("Square Numbers", fontsize=24)
plt.xlabel("Value", fontsize=18)
plt.ylabel("Square of Value", fontsize=18)
plt.tick_params(axis='both', which='major',
                labelsize=14)
plt.axis([0, 1100, 0, 1100000])

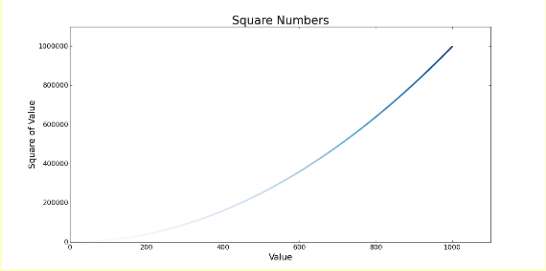
plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the `c` *argument, and the* `cmap` *argument specifies which colormap to use.*

The `edgecolor='none'` *argument removes the black outline from each point.*

```
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)
```



Customizing plots (cont.)

Emphasizing points

You can plot as much data as you want on one plot. Here we re-plot the first and last points larger to emphasize them.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)

plt.scatter(x_values[0], squares[0], c='green',
        edgecolor='none', s=100)
plt.scatter(x_values[-1], squares[-1], c='red',
        edgecolor='none', s=100)

plt.title("Square Numbers", fontsize=24)
--snip--
```

Removing axes

You can customize or remove axes entirely. Here's how to access each axis, and hide it.

```
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

Setting a custom figure size

You can make your plot as big or small as you want. Before plotting your data, add the following code. The `dpi` *argument is optional; if you don't know your system's resolution you can omit the argument and adjust the* `figsize` *argument accordingly.*

```
plt.figure(dpi=128, figsize=(10, 6))
```

Saving a plot

The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace `plt.show()` *with* `plt.savefig()`*. The* `bbox_inches='tight'` *argument trims extra whitespace from the plot.*

```
plt.savefig('squares.png', bbox_inches='tight')
```

Online resources

The matplotlib gallery and documentation are at http://matplotlib.org/. Be sure to visit the examples, gallery, and pyplot links.

Multiple plots

You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.

Plotting two sets of data

Here we use `plt.scatter()` twice to plot square numbers and cubes on the same figure.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.scatter(x_values, squares, c='blue',
        edgecolor='none', s=20)
plt.scatter(x_values, cubes, c='red',
        edgecolor='none', s=20)

plt.axis([0, 11, 0, 1100])
plt.show()
```

Filling the space between data sets

The `fill_between()` method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a `facecolor` to use for the fill, and an optional `alpha` argument that controls the color's transparency.

```
plt.fill_between(x_values, cubes, squares,
        facecolor='blue', alpha=0.25)
```

Working with dates and times

Many interesting data sets have a date or time as the x-value. Python's datetime module helps you work with this kind of data.

Generating the current date

The `datetime.now()` function returns a datetime object representing the current date and time.

```
from datetime import datetime as dt

today = dt.now()
date_string = dt.strftime(today, '%m/%d/%Y')
print(date_string)
```

Generating a specific date

You can also generate a datetime object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.

```
from datetime import datetime as dt

new_years = dt(2017, 1, 1)
fall_equinox = dt(year=2016, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments

The `strftime()` function generates a formatted string from a datetime object, and the `strptime()` function genereates a datetime object from a string. The following codes let you work with dates exactly as you need to.

```
%A    Weekday name, such as Monday
%B    Month name, such as January
%m    Month, as a number (01 to 12)
%d    Day of the month, as a number (01 to 31)
%Y    Four-digit year, such as 2016
%y    Two-digit year, such as 16
%H    Hour, in 24-hour format (00 to 23)
%I    Hour, in 12-hour format (01 to 12)
%p    AM or PM
%M    Minutes (00 to 59)
%S    Seconds (00 to 61)
```

Converting a string to a datetime object

```
new_years = dt.strptime('1/1/2017', '%m/%d/%Y')
```

Converting a datetime object to a string

```
ny_string = dt.strftime(new_years, '%B %d, %Y')
print(ny_string)
```

Plotting high temperatures

The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2016, 6, 21), dt(2016, 6, 22),
    dt(2016, 6, 23), dt(2016, 6, 24),
    ]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6))
plt.plot(dates, highs, c='red')
plt.title("Daily High Temps", fontsize=24)
plt.ylabel("Temp (F)", fontsize=16)

x_axis = plt.axes().get_xaxis()
x_axis.set_major_formatter(
        mdates.DateFormatter('%B %d %Y')
        )
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.

Sharing an x-axis

The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis.

The `plt.subplots()` function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.

```
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(2, 1, sharex=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```

Sharing a y-axis

To share a y-axis, we use the `sharey=True` argument.

```
import matplotlib.pyplot as plt

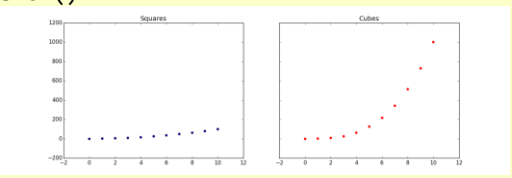
x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(1, 2, sharey=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```



More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Pygal

What is Pygal?

Data visualization involves exploring data through visual representations. Pygal helps you make visually appealing representations of the data you're working with. Pygal is particularly well suited for visualizations that will be presented online, because it supports interactive elements.

Installing Pygal

Pygal can be installed using pip.

Pygal on Linux and OS X

```
$ pip install --user pygal
```

Pygal on Windows

```
> python -m pip install --user pygal
```

Line graphs, scatter plots, and bar graphs

To make a plot with Pygal, you specify the kind of plot and then add the data.

Making a line graph

To view the output, open the file squares.svg in a browser.

```
import pygal

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Adding labels and a title

```
--snip--
chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares"
chart.x_labels = x_values
chart.x_title = "Value"
chart.y_title = "Square of Value"
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Line graphs, scatter plots, and bar graphs (cont.)

Making a scatter plot

The data for a scatter plot needs to be a list containing tuples of the form (x, y). The `stroke=False` argument tells Pygal to make an XY chart with no line connecting the points.

```
import pygal

squares = [
    (0, 0), (1, 1), (2, 4), (3, 9),
    (4, 16), (5, 25),
    ]

chart = pygal.XY(stroke=False)
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Using a list comprehension for a scatter plot

A list comprehension can be used to effficiently make a dataset for a scatter plot.

```
squares = [(x, x**2) for x in range(1000)]
```

Making a bar graph

A bar graph requires a list of values for the bar sizes. To label the bars, pass a list of the same length to `x_labels`.

```
import pygal

outcomes = [1, 2, 3, 4, 5, 6]
frequencies = [18, 16, 18, 17, 18, 13]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = outcomes
chart.add('D6', frequencies)
chart.render_to_file('rolling_dice.svg')
```

Making a bar graph from a dictionary

Since each bar needs a label and a value, a dictionary is a great way to store the data for a bar graph. The keys are used as the labels along the x-axis, and the values are used to determine the height of each bar.

```
import pygal

results = {
    1:18, 2:16, 3:18,
    4:17, 5:18, 6:13,
    }

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = results.keys()
chart.add('D6', results.values())
chart.render_to_file('rolling_dice.svg')
```

Multiple plots

You can add as much data as you want when making a visualization.

Plotting squares and cubes

```
import pygal

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Filling the area under a data series

Pygal allows you to fill the area under or over each series of data. The default is to fill from the x-axis up, but you can fill from any horizontal line using the `zero` argument.

```
chart = pygal.Line(fill=True, zero=0)
```

Online resources

The documentation for Pygal is available at http://www.pygal.org/.

Enabling interactive features

If you're viewing svg output in a browser, Pygal needs to render the output file in a specific way. The `force_uri_protocol` attribute for chart objects needs to be set to `'http'`.

Styling plots

Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.

Using built-in styles

To use built-in styles, import the style and make an instance of the style class. Then pass the style object with the `style` argument when you make the chart object.

```
import pygal
from pygal.style import LightGreenStyle

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart_style = LightGreenStyle()
chart = pygal.Line(style=chart_style)
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Parametric built-in styles

Some built-in styles accept a custom color, then generate a theme based on that color.

```
from pygal.style import LightenStyle

--snip--
chart_style = LightenStyle('#336688')
chart = pygal.Line(style=chart_style)
--snip--
```

Customizing individual style properties

Style objects have a number of properties you can set individually.

```
chart_style = LightenStyle('#336688')
chart_style.plot_background = '#CCCCCC'
chart_style.major_label_font_size = 20
chart_style.label_font_size = 16
--snip--
```

Custom style class

You can start with a bare style class, and then set only the properties you care about.

```
chart_style = Style()
chart_style.colors = [
    '#CCCCCC', '#AAAAAA', '#888888']
chart_style.plot_background = '#EEEEEE'

chart = pygal.Line(style=chart_style)
--snip--
```

Styling plots (cont.)

Configuration settings

Some settings are controlled by a `Config` object.

```
my_config = pygal.Config()
my_config.show_y_guides = False
my_config.width = 1000
my_config.dots_size = 5

chart = pygal.Line(config=my_config)
--snip--
```

Styling series

You can give each series on a chart different style settings.

```
chart.add('Squares', squares, dots_size=2)
chart.add('Cubes', cubes, dots_size=3)
```

Styling individual data points

You can style individual data points as well. To do so, write a dictionary for each data point you want to customize. A `'value'` key is required, and other properies are optional.

```
import pygal

repos = [
    {
        'value': 20506,
        'color': '#3333CC',
        'xlink': 'http://djangoproject.com/',
    },
    20054,
    12607,
    11827,
    ]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = [
    'django', 'requests', 'scikit-learn',
    'tornado',
    ]
chart.y_title = 'Stars'
chart.add('Python Repos', repos)
chart.render_to_file('python_repos.svg')
```

Plotting global datasets

Pygal can generate world maps, and you can add any data you want to these maps. Data is indicated by coloring, by labels, and by tooltips that show data when users hover over each country on the map.

Installing the world map module

The world map module is not included by default in Pygal 2.0. It can be installed with pip:

```
$ pip install --user pygal_maps_world
```

Making a world map

The following code makes a simple world map showing the countries of North America.

```
from pygal.maps.world import World

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'North America'
wm.add('North America', ['ca', 'mx', 'us'])

wm.render_to_file('north_america.svg')
```

Showing all the country codes

In order to make maps, you need to know Pygal's country codes. The following example will print an alphabetical list of each country and its code.

```
from pygal.maps.world import COUNTRIES

for code in sorted(COUNTRIES.keys()):
    print(code, COUNTRIES[code])
```

Plotting numerical data on a world map

To plot numerical data on a map, pass a dictionary to `add()` instead of a list.

```
from pygal.maps.world import World

populations = {
    'ca': 34126000,
    'us': 309349000,
    'mx': 113423000,
    }

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'Population of North America'
wm.add('North America', populations)

wm.render_to_file('na_populations.svg')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django

What is Django?

Django is a web framework which helps you build interactive websites using Python. With Django you define the kind of data your site needs to work with, and you define the ways your users can work with that data.

Installing Django

It's usualy best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you're working in an active virtual environment.

Create a virtual environment

```
$ python -m venv ll_env
```

Activate the environment (Linux and OS X)

```
$ source ll_env/bin/activate
```

Activate the environment (Windows)

```
> ll_env\Scripts\activate
```

Install Django to the active environment

```
(ll_env)$ pip install Django
```

Creating a project

To start a project we'll create a new project, create a database, and start a development server.

Create a new project

```
$ django-admin.py startproject learning_log .
```

Create a database

```
$ python manage.py migrate
```

View the project

After issuing this command, you can view the project at http://localhost:8000/.

```
$ python manage.py runserver
```

Create a new app

A Django project is made up of one or more apps.

```
$ python manage.py startapp learning_logs
```

Working with models

The data in a Django project is structured as a set of models.

Defining a model

To define the models for your app, modify the file models.py that was created in your app's folder. The __str__() method tells Django how to represent data objects based on this model.

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

Activating a model

To use a model the app must be added to the tuple `INSTALLED_APPS`*, which is stored in the project's settings.py file.*

```
INSTALLED_APPS = (
    --snip--
    'django.contrib.staticfiles',

    # My apps
    'learning_logs',
    )
```

Migrating the database

The database needs to be modified to store the kind of data that the model represents.

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

Creating a superuser

A superuser is a user account that has access to all aspects of the project.

```
$ python manage.py createsuperuser
```

Registering a model

You can register your models with Django's admin site, which makes it easier to work with the data in your project. To do this, modify the app's admin.py file. View the admin site at http://localhost:8000/admin/.

```
from django.contrib import admin

from learning_logs.models import Topic

admin.site.register(Topic)
```

Building a simple home page

Users interact with a project through web pages, and a project's home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.

Mapping a project's URLs

The project's main urls.py file tells Django where to find the urls.py files associated with each app in the project.

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Mapping an app's URLs

An app's urls.py file tells Django which view to use for each URL in the app. You'll need to make this file yourself, and save it in the app's folder.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    ]
```

Writing a simple view

A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app's views.py file. This simple view function doesn't pull in any data, but it uses the template index.html to render the home page.

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request,
            'learning_logs/index.html')
```

Online resources

The documentation for Django is available at http://docs.djangoproject.com/. The Django documentation is thorough and user-friendly, so check it out!

Building a simple home page (cont.)

Writing a simple template

A template sets up the structure for a page. It's a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your
learning, for any topic you're learning
about.</p>
```

Template inheritance

Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site.

The parent template

The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
</p>

{% block content %}{% endblock content %}
```

The child template

The child template uses the `{% extends %}` template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.

```
{% extends 'learning_logs/base.html' %}

{% block content %}
  <p>
    Learning Log helps you keep track
    of your learning, for any topic you're
    learning about.
  </p>
{% endblock content %}
```

Template indentation

Python code is usually indented by four spaces. In templates you'll often see two spaces used for indentation, because elements tend to be nested more deeply in templates.

Another model

A new model can use an existing model. The ForeignKey attribute establishes a connection between instances of the two related models. Make sure to migrate the database after adding a new model to your app.

Defining a model with a foreign key

```
class Entry(models.Model):
    """Learning log entries for a topic."""
    topic = models.ForeignKey(Topic)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text[:50] + "..."
```

Building a page with data

Most pages in a project need to present data that's specific to the current user.

URL parameters

A URL often needs to accept a parameter telling it which data to access from the database. The second URL pattern shown here looks for the ID of a specific topic and stores it in the parameter `topic_id`.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^topics/(?P<topic_id>\d+)/$',
            views.topic, name='topic'),
    ]
```

Using data in a view

The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a `context` dictionary to the template, containing data that should be displayed on the page.

```
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
        }
    return render(request,
        'learning_logs/topic.html', context)
```

Restarting the development server

If you make a change to your project and the change doesn't seem to have any effect, try restarting the server:

```
$ python manage.py runserver
```

Building a page with data (cont.)

Using data in a template

The data in the view function's `context` dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces.

The vertical line after a template variable indicates a filter. In this case a filter called `date` formats date objects, and the filter `linebreaks` renders paragraphs properly on a web page.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <ul>
  {% for entry in entries %}
    <li>
    <p>
      {{ entry.date_added|date:'M d, Y H:i' }}
    </p>
    <p>
      {{ entry.text|linebreaks }}
    </p>
    </li>
  {% empty %}
    <li>There are no entries yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

The Django shell

You can explore the data in your project from the command line. This is helpful for developing queries and testing code snippets.

Start a shell session

```
$ python manage.py shell
```

Access data from the project

```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django, Part 2

Users and forms

Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and some may be public. Django's forms allow users to enter and modify their data.

User accounts

User accounts are handled by a dedicated app called `users`*. Users need to be able to register, log in, and log out. Django automates much of this work for you.*

Making a users app

After making the app, be sure to add `'users'` *to* `INSTALLED_APPS` *in the project's settings.py file.*

```
$ python manage.py startapp users
```

Including URLS for the users app

Add a line to the project's urls.py file so the `users` *app's URLs are included in the project.*

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^users/', include('users.urls',
            namespace='users')),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Using forms in Django

There are a number of ways to create forms and work with them. You can use Django's defaults, or completely customize your forms. For a simple way to let users enter data based on your models, use a `ModelForm`*. This creates a form that allows users to enter data that will populate the fields on a model.*

The register view on the back of this sheet shows a simple approach to form processing. If the view doesn't receive data from a form, it responds with a blank form. If it receives `POST` *data from a form, it validates the data and then saves it to the database.*

User accounts (cont.)

Defining the URLs

Users will need to be able to log in, log out, and register. Make a new urls.py file in the `users` *app folder. The login view is a default view provided by Django.*

```
from django.conf.urls import url
from django.contrib.auth.views import login

from . import views

urlpatterns = [
    url(r'^login/$', login,
        {'template_name': 'users/login.html'},
        name='login'),
    url(r'^logout/$', views.logout_view,
        name='logout'),
    url(r'^register/$', views.register,
        name='register'),
    ]
```

The login template

The login view is provided by default, but you need to provide your own login template. The template shown here displays a simple login form, and provides basic error messages. Make a templates folder in the users folder, and then make a users folder in the templates folder. Save this file as login.html.

The tag `{% csrf_token %}` *helps prevent a common type of attack with forms. The* `{{ form.as_p }}` *element displays the default login form in paragraph format. The* `<input>` *element named* `next` *redirects the user to the home page after a successful login.*

```
{% extends "learning_logs/base.html" %}

{% block content %}
  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form method="post"
      action="{% url 'users:login' %}">
    {% csrf token %}
    {{ form.as_p }}
    <button name="submit">log in</button>

    <input type="hidden" name="next"
      value="{% url 'learning_logs:index' %}"/>
  </form>

{% endblock content %}
```

User accounts (cont.)

Showing the current login status

You can modify the base.html template to show whether the user is currently logged in, and to provide a link to the login and logout pages. Django makes a `user` *object available to every template, and this template takes advantage of this object.*

The `user.is_authenticated` *tag allows you to serve specific content to users depending on whether they have logged in or not. The* `{{ user.username }}` *property allows you to greet users who have logged in. Users who haven't logged in see links to register or log in.*

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">
      log out
    </a>
  {% else %}
    <a href="{% url 'users:register' %}">
      register
    </a> -
    <a href="{% url 'users:login' %}">
      log in
    </a>
  {% endif %}
</p>

{% block content %}{% endblock content %}
```

The logout view

The `logout_view()` *function uses Django's* `logout()` *function and then redirects the user back to the home page. Since there is no logout page, there is no logout template. Make sure to write this code in the views.py file that's stored in the users app folder.*

```
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.contrib.auth import logout

def logout_view(request):
    """Log the user out."""
    logout(request)
    return HttpResponseRedirect(
        reverse('learning_logs:index'))
```

User accounts (cont.)

The register view

The register view needs to display a blank registration form when the page is first requested, and then process completed registration forms. A successful registration logs the user in and redirects to the home page.

```
from django.contrib.auth import login
from django.contrib.auth import authenticate
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Show blank registration form.
        form = UserCreationForm()
    else:
        # Process completed form.
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()
            # Log in, redirect to home page.
            pw = request.POST['password1']
            authenticated_user = authenticate(
                username=new_user.username,
                password=pw
            )
            login(request, authenticated_user)
            return HttpResponseRedirect(
                reverse('learning_logs:index'))

    context = {'form': form}
    return render(request,
        'users/register.html', context)
```

Styling your project

The django-bootstrap3 app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at http://django-bootstrap3.readthedocs.io/.

Deploying your project

Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment. You'll need to install a set of heroku tools, and use git to track the state of your project. See http://devcenter.heroku.com/, and click on the Python link.

User accounts (cont.)

The register template

The register template displays the registration form in paragraph formats.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <form method='post'
      action="{% url 'users:register' %}">

    {% csrf_token %}
    {{ form.as_p }}

    <button name='submit'>register</button>
    <input type='hidden' name='next'
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

Connecting data to users

Users will have data that belongs to them. Any model that should be connected directly to a user needs a field connecting instances of the model to a specific user.

Making a topic belong to a user

Only the highest-level data in a hierarchy needs to be directly connected to a user. To do this import the `User` model, and add it as a foreign key on the data model.

After modifying the model you'll need to migrate the database. You'll need to choose a user ID to connect each existing instance to.

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)
    owner = models.ForeignKey(User)

    def __str__(self):
        return self.text
```

Querying data for the current user

In a view, the request object has a `user` attribute. You can use this attribute to query for the user's data. The `filter()` function then pulls the data that belongs to the current user.

```
topics = Topic.objects.filter(
                owner=request.user)
```

Connecting data to users (cont.)

Restricting access to logged-in users

Some pages are only relevant to registered users. The views for these pages can be protected by the `@login_required` decorator. Any view with this decorator will automatically redirect non-logged in users to an appropriate page. Here's an example views.py file.

```
from django.contrib.auth.decorators import /
        login_required
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
```

Setting the redirect URL

The `@login_required` decorator sends unauthorized users to the login page. Add the following line to your project's settings.py file so Django will know how to find your login page.

```
LOGIN_URL = '/users/login/'
```

Preventing inadvertent access

Some pages serve data based on a parameter in the URL. You can check that the current user owns the requested data, and return a 404 error if they don't. Here's an example view.

```
from django.http import Http404

--snip--
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --snip--
```

Using a form to edit data

If you provide some initial data, Django generates a form with the user's existing data. Users can then modify and save their data.

Creating a form with initial data

The instance parameter allows you to specify initial data for a form.

```
form = EntryForm(instance=entry)
```

Modifying data before saving

The argument `commit=False` allows you to make changes before writing data to the database.

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Essential Cheat Sheets for Machine Learning and Deep Learning Engineers

Kailash Ahirwar · Follow
Published in Startups & Venture Capital · 4 min read · May 28, 2017

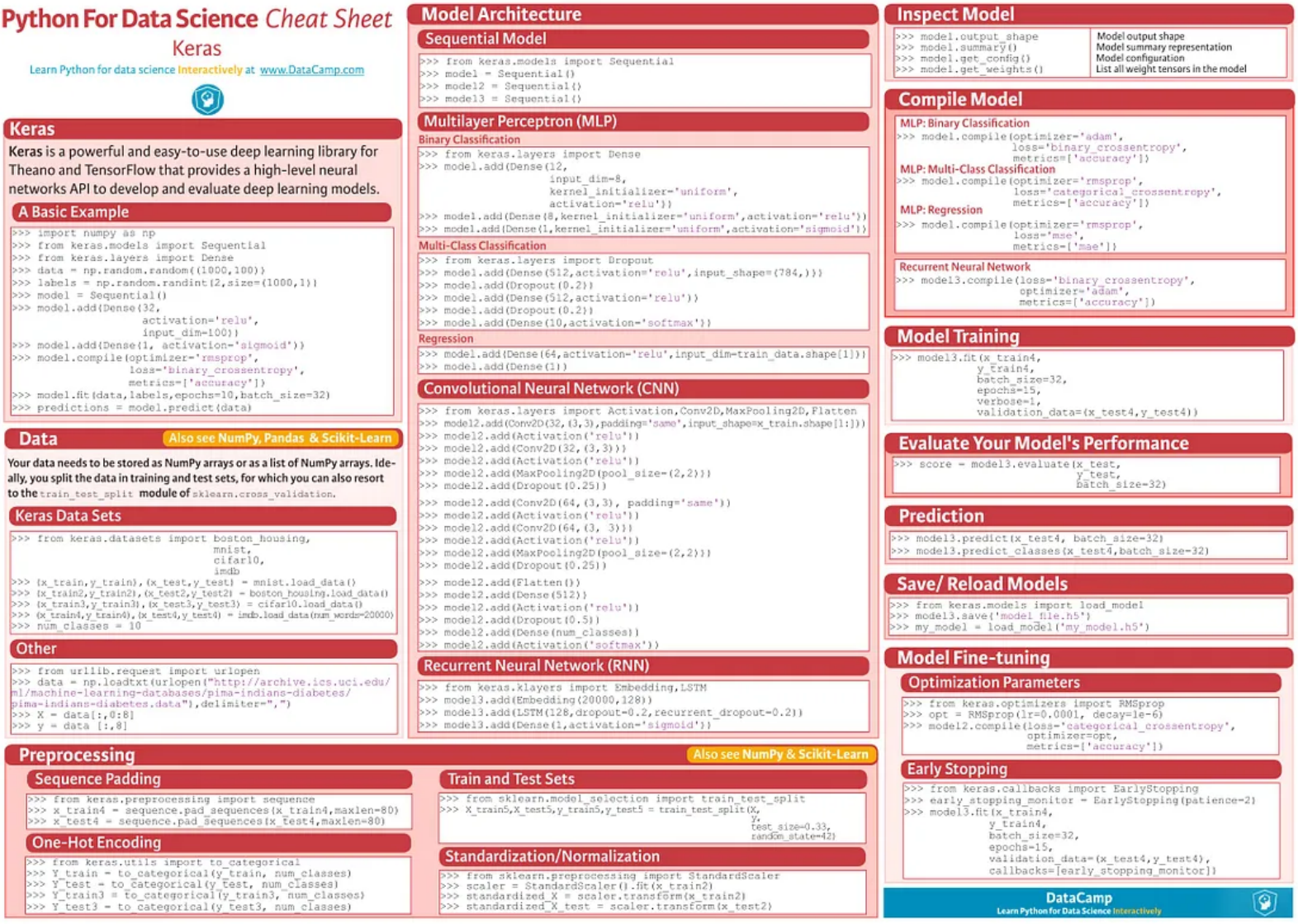
Machine learning is complex. For newbies, starting to learn machine learning can be painful if they don't have right resources to learn from. Most of the machine learning libraries are difficult to understand and learning curve can be a bit frustrating. I am creating a repository on Github(cheatsheets-ai) containing cheatsheets for different machine learning frameworks, gathered from different sources. Do visit the Github repository, also, contribute cheat sheets if you have any. Thanks.

List of Cheatsheets:

1. Keras
2. Numpy
3. Pandas
4. Scipy
5. Matplotlib
6. Scikit-learn

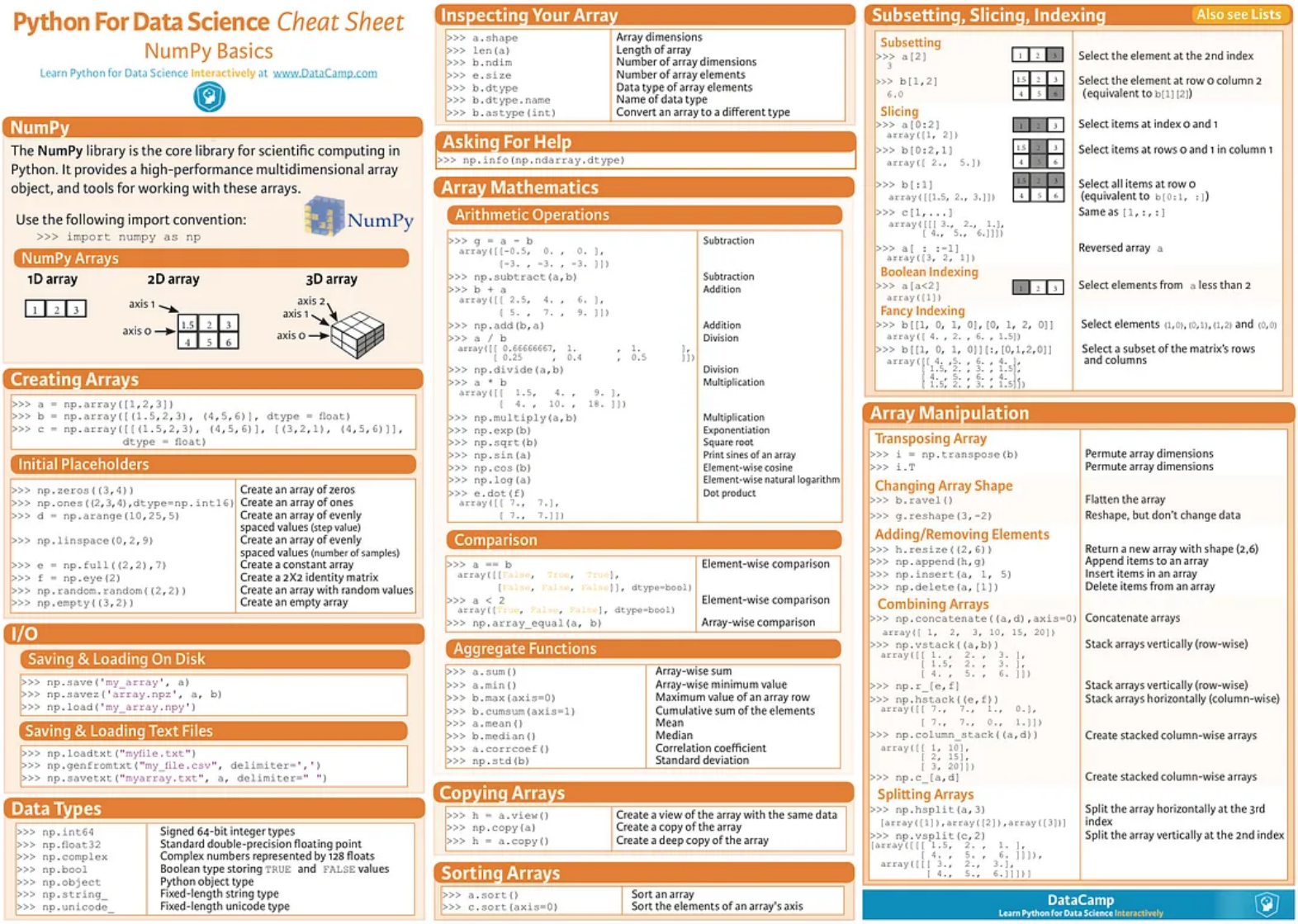
7. Neural Networks Zoo
8. ggplot2
9. PySpark
10. R Studio
11. Jupyter Notebook
12. Dask

1. Keras



Source — https://www.datacamp.com/community/blog/keras-cheat-sheet#gs.DRKeNMs

2. Numpy



Source — https://www.datacamp.com/community/blog/python-numpy-cheat-sheet#gs.AK5ZBgE

3. Pandas

Data Wrangling with pandas Cheat Sheet

http://pandas.pydata.org

Tidy Data – A foundation for wrangling in pandas

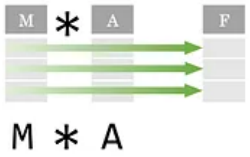
In a tidy data set:

&

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Syntax – Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```

Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```

Specify values for each row.

		a	b	c
n	v			
d	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d',1),('d',2),('e',2)],
          names=['n','v'])))
```

Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
        .rename(columns={
                'variable' : 'var',
                'value' : 'val'})
        .query('val >= 200')
    )
```

Reshaping Data – Change the layout of a data set

`pd.melt(df)`
Gather columns into rows.

`df.pivot(columns='var', values='val')`
Spread rows into columns.

`pd.concat([df1,df2])`
Append rows of DataFrames

`pd.concat([df1,df2], axis=1)`
Append columns of DataFrames

`df.sort_values('mpg')`
Order rows by values of a column (low to high).

`df.sort_values('mpg',ascending=False)`
Order rows by values of a column (high to low).

`df.rename(columns = {'y':'year'})`
Rename the columns of a DataFrame

`df.sort_index()`
Sort the index of a DataFrame

`df.reset_index()`
Reset index of DataFrame to row numbers, moving index to columns.

`df.drop(['Length','Height'], axis=1)`
Drop columns from DataFrame

Subset Observations (Rows)

`df[df.Length > 7]`
Extract rows that meet logical criteria.

`df.drop_duplicates()`
Remove duplicate rows (only considers columns).

`df.head(n)`
Select first n rows.

`df.tail(n)`
Select last n rows.

`df.sample(frac=0.5)`
Randomly select fraction of rows.

`df.sample(n=10)`
Randomly select n rows.

`df.iloc[10:20]`
Select rows by position.

`df.nlargest(n, 'value')`
Select and order top n entries.

`df.nsmallest(n, 'value')`
Select and order bottom n entries.

Logic in Python (and pandas)			
<	Less than	!=	Not equal to
>	Greater than	`df.column.isin(values)`	Group membership
==	Equals	`pd.isnull(obj)`	Is NaN
<=	Less than or equals	`pd.notnull(obj)`	Is not NaN
>=	Greater than or equals	`&,|,~,^,df.any(),df.all()`	Logical and, or, not, xor, any, all

Subset Variables (Columns)

`df[['width','length','species']]`
Select multiple columns with specific names.

`df['width']` *or* `df.width`
Select single column with specific name.

`df.filter(regex='regex')`
Select columns whose name matches regular expression *regex*.

regex (Regular Expressions) Examples	
`'\.'`	Matches strings containing a period '.'
`'Length$'`	Matches strings ending with word 'Length'
`'^Sepal'`	Matches strings beginning with the word 'Sepal'
`'^x[1-5]$'`	Matches strings beginning with 'x' and ending with 1,2,3,4,5
`'^(?!Species$).*'`	Matches strings except the string 'Species'

`df.loc[:,'x2':'x4']`
Select all columns between x2 and x4 (inclusive).

`df.iloc[:,[1,2,5]]`
Select columns in positions 1, 2 and 5 (first column is 0).

`df.loc[df['a'] > 10, ['a','c']]`
Select rows meeting logical condition, and only the specific columns .

Summarize Data

`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive statistics for each column (or GroupBy)

pandas provides a large set of **summary functions** that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

`std()`
Standard deviation of each object.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

`df.fillna(value)`
Replace all NA/null data with value.

Make New Columns

`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.

pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

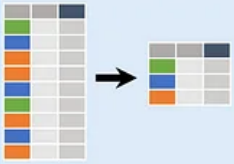
`max(axis=1)`
Element-wise max.

`min(axis=1)`
Element-wise min.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

`abs()`
Absolute value.

Group Data



`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()`
Size of each group.

`agg(function)`
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`shift(-1)`
Copy with values lagged by 1.

`rank(method='dense')`
Ranks with no gaps.

`cumsum()`
Cumulative sum.

`rank(method='min')`
Ranks. Ties get min rank.

`cummax()`
Cumulative max.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`cummin()`
Cumulative min.

`rank(method='first')`
Ranks. Ties go to first value.

`cumprod()`
Cumulative product.

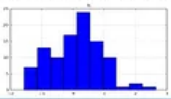
Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

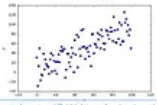
`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Plotting

`df.plot.hist()`
Histogram for each column



`df.plot.scatter(x='w',y='h')`
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

`pd.merge(adf, bdf, how='left', on='x1')`
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

`pd.merge(adf, bdf, how='right', on='x1')`
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

`pd.merge(adf, bdf, how='inner', on='x1')`
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

`pd.merge(adf, bdf, how='outer', on='x1')`
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(['_merge'],axis=1)
```
Rows that appear in ydf but not zdf (Setdiff).

http://pandas.pydata.org/ This cheat sheet inspired by Rstudio Data Wrangling Cheatsheet (https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf) Written by Irv Lustig, Princeton Consultants

Source — https://www.datacamp.com/community/blog/pandas-cheat-sheet-python#gs.HPFoRIc

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science Interactively at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

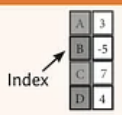
Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

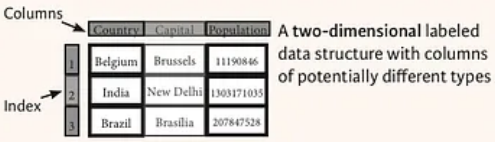
Series

A **one-dimensional** labeled array capable of holding any data type




```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
-5
```
Get one element

```
>>> df[1:]
   Country     Capital  Population
1   India   New Delhi  1303171035
2  Brazil    Brasilia   207847528
```
Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]]
'Belgium'
>>> df.iat[[0],[0]]
'Belgium'
```
Select single value by row & column

By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'
>>> df.at[[0], ['Country']]
'Belgium'
```
Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
Country       Brazil
Capital     Brasilia
Population  207847528
```
Select single row of subset of rows

```
>>> df.ix[:,'Capital']
0     Brussels
1    New Delhi
2     Brasilia
```
Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
'New Delhi'
```
Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population']>1200000000]
```
Series `s` where value is not >1
`s` where value is <-1 or >2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```
Set index `a` of Series `s` to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```
Drop values from rows (axis=0)
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index(by='Country')
>>> s.order()
>>> df.rank()
```
Sort by row or column index
Sort a series by its values
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```
(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idmin()/df.idmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```
Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```
Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
  a     10.0
  b      NaN
  c      5.0
  d      7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
  a    10.0
  b    -5.0
  c    5.0
  d    7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> pd.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

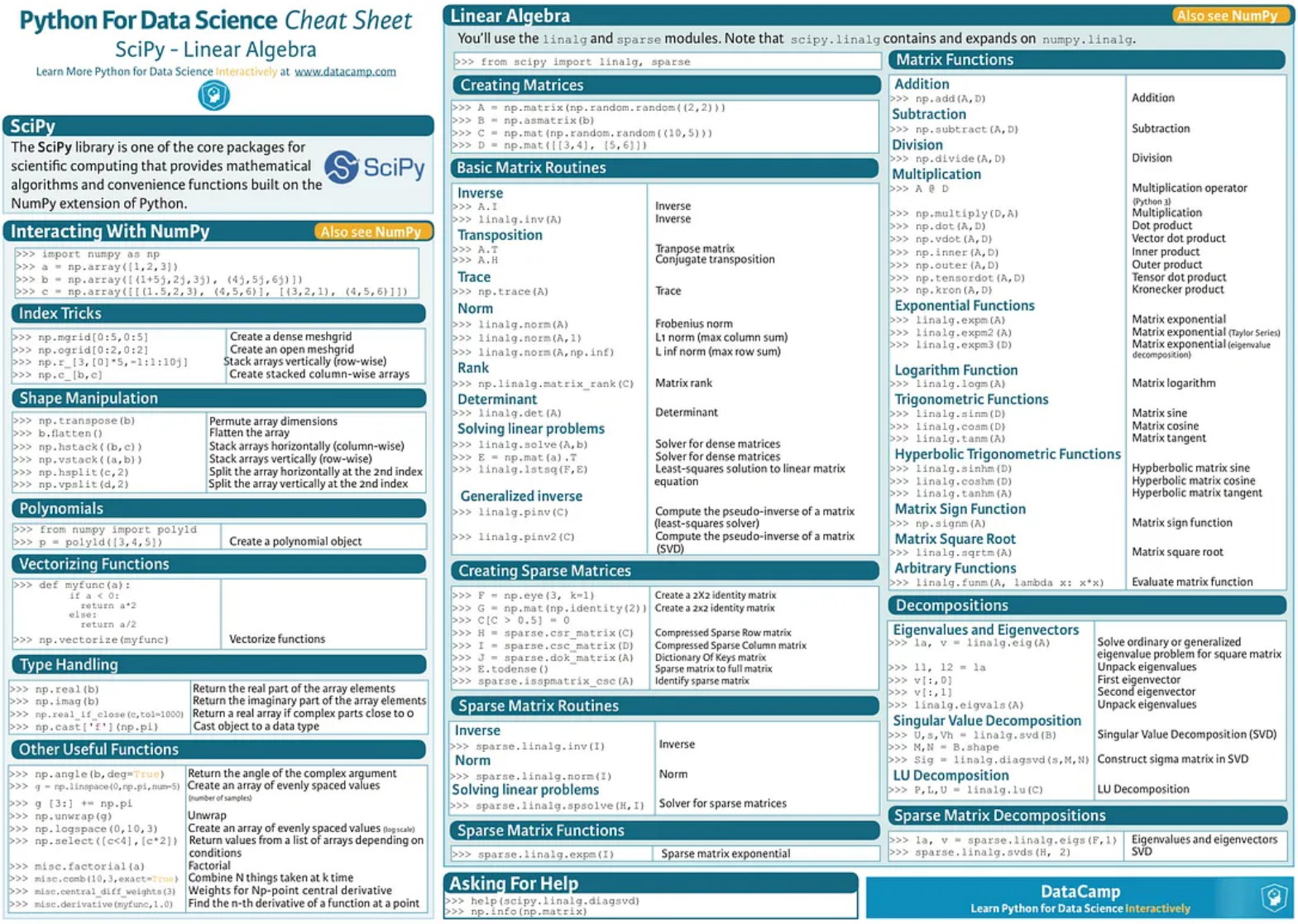
`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> pd.to_sql('myDf', engine)
```

DataCamp
Learn Python for Data Science Interactively

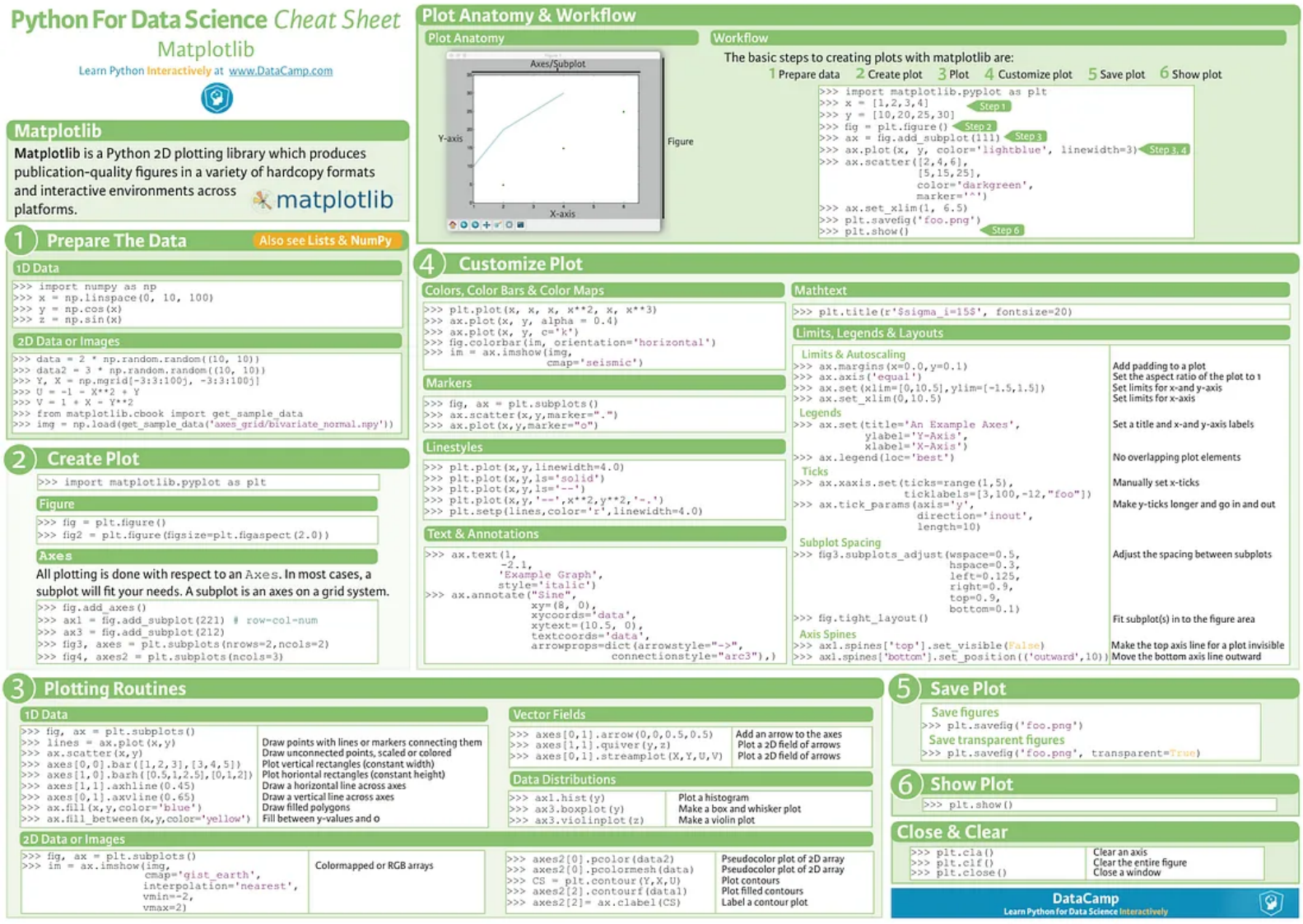
Source — https://www.datacamp.com/community/blog/python-pandas-cheat-sheet#gs.oundfxM

4. Scipy



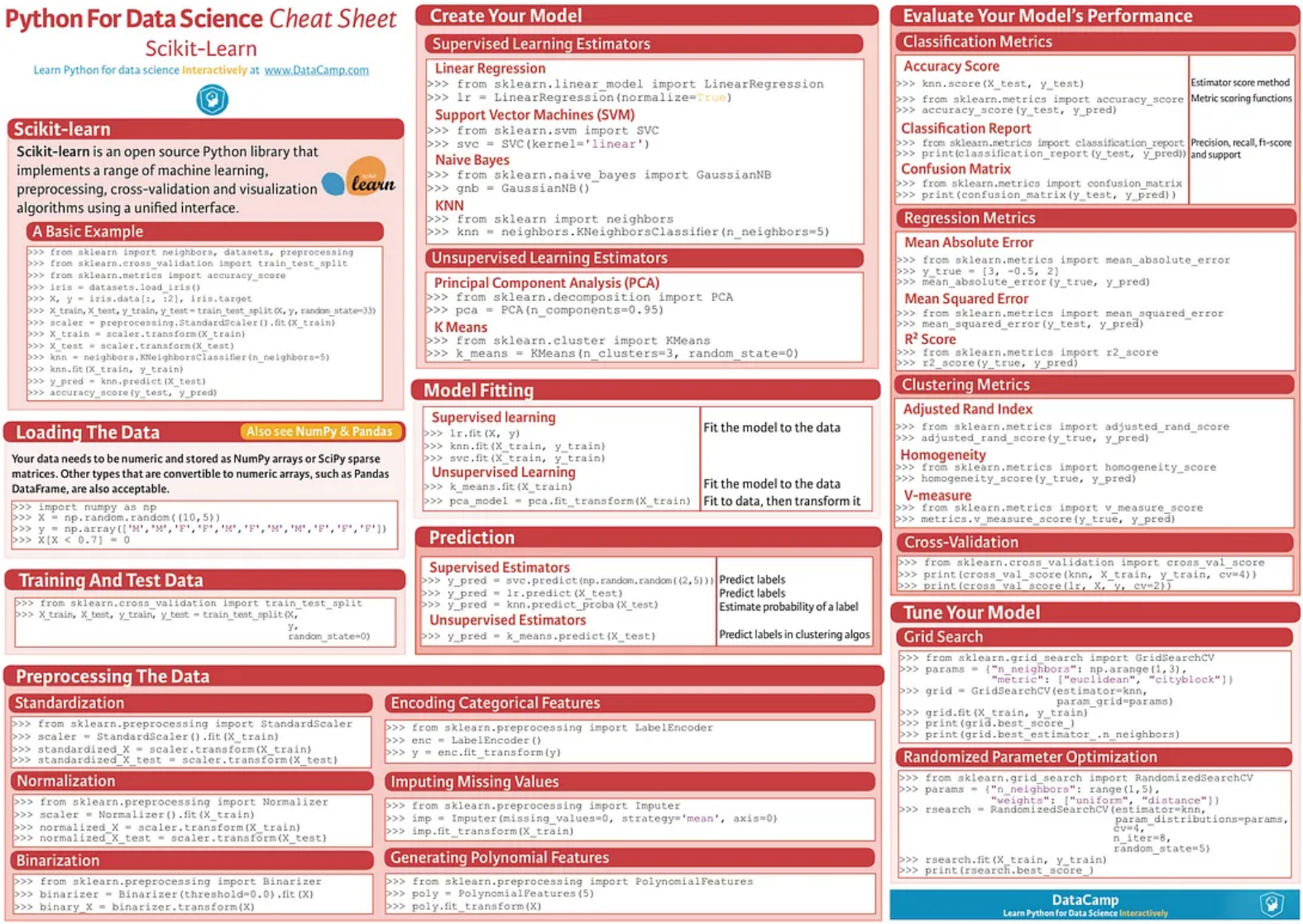
Source — https://www.datacamp.com/community/blog/python-scipy-cheat-sheet#gs.JDSg3OI

5. Matplotlib



Source — https://www.datacamp.com/community/blog/python-matplotlib-cheat-sheet#gs.uEKySpY

6. Scikit-learn



Source — https://www.datacamp.com/community/blog/scikit-learn-cheat-sheet

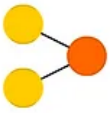
7. Neural Networks Zoo

A mostly complete chart of

Neural Networks

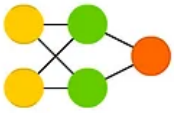
- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Perceptron (P)

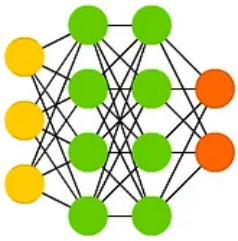


Feed Forward (FF)

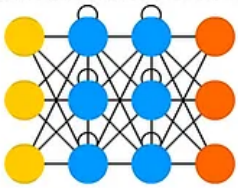
Radial Basis Network (RBF)



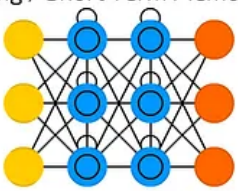
Deep Feed Forward (DFF)



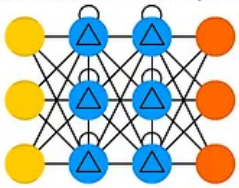
Recurrent Neural Network (RNN)



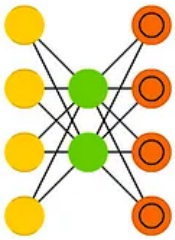
Long / Short Term Memory (LSTM)



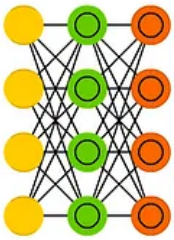
Gated Recurrent Unit (GRU)



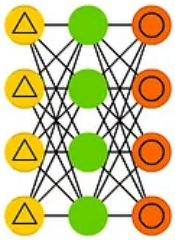
Auto Encoder (AE)



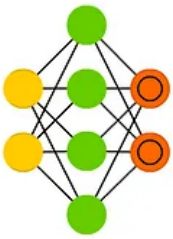
Variational AE (VAE)



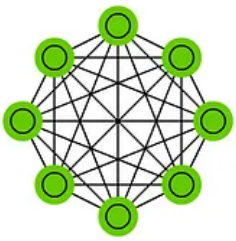
Denoising AE (DAE)



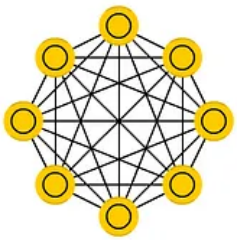
Sparse AE (SAE)



Markov Chain (MC)

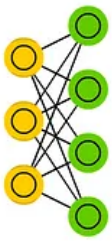


Hopfield Network (HN)

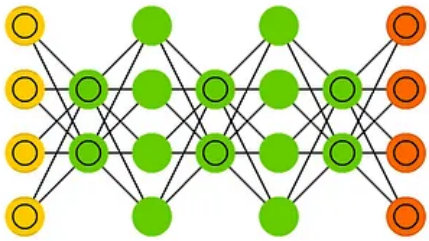


Boltzmann Machine (BM)

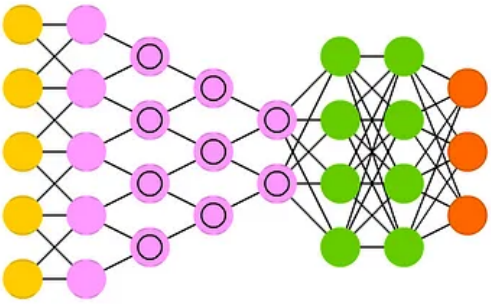
Restricted BM (RBM)



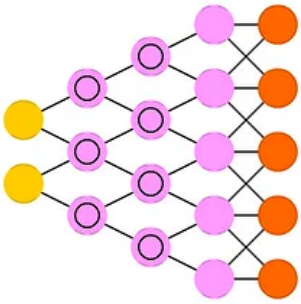
Deep Belief Network (DBN)



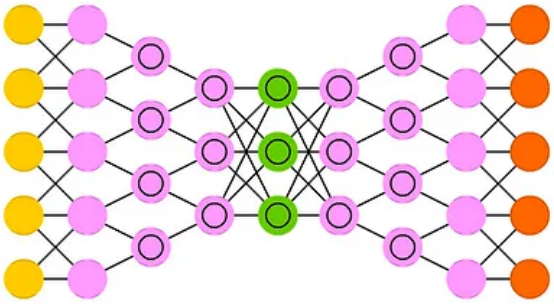
Deep Convolutional Network (DCN)



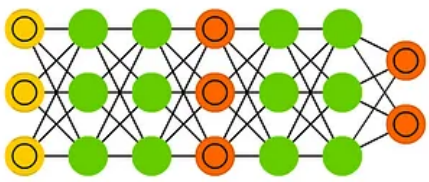
Deconvolutional Network (DN)



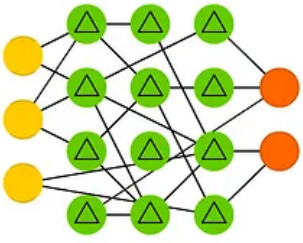
Deep Convolutional Inverse Graphics Network (DCIGN)



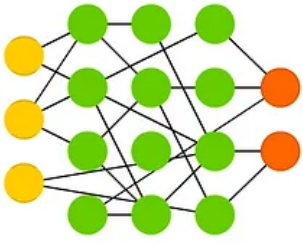
Generative Adversarial Network (GAN)



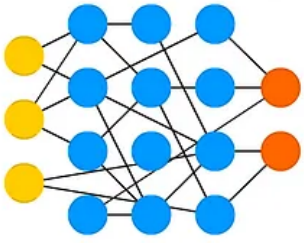
Liquid State Machine (LSM)



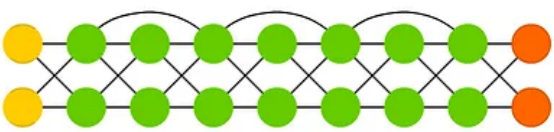
Extreme Learning Machine (ELM)



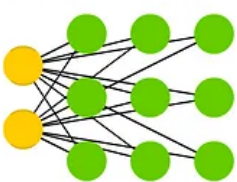
Echo State Network (ESN)



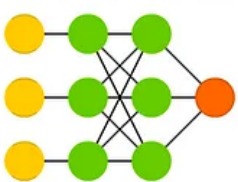
Deep Residual Network (DRN)



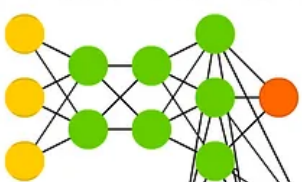
Kohonen Network (KN)



Support Vector Machine (SVM)

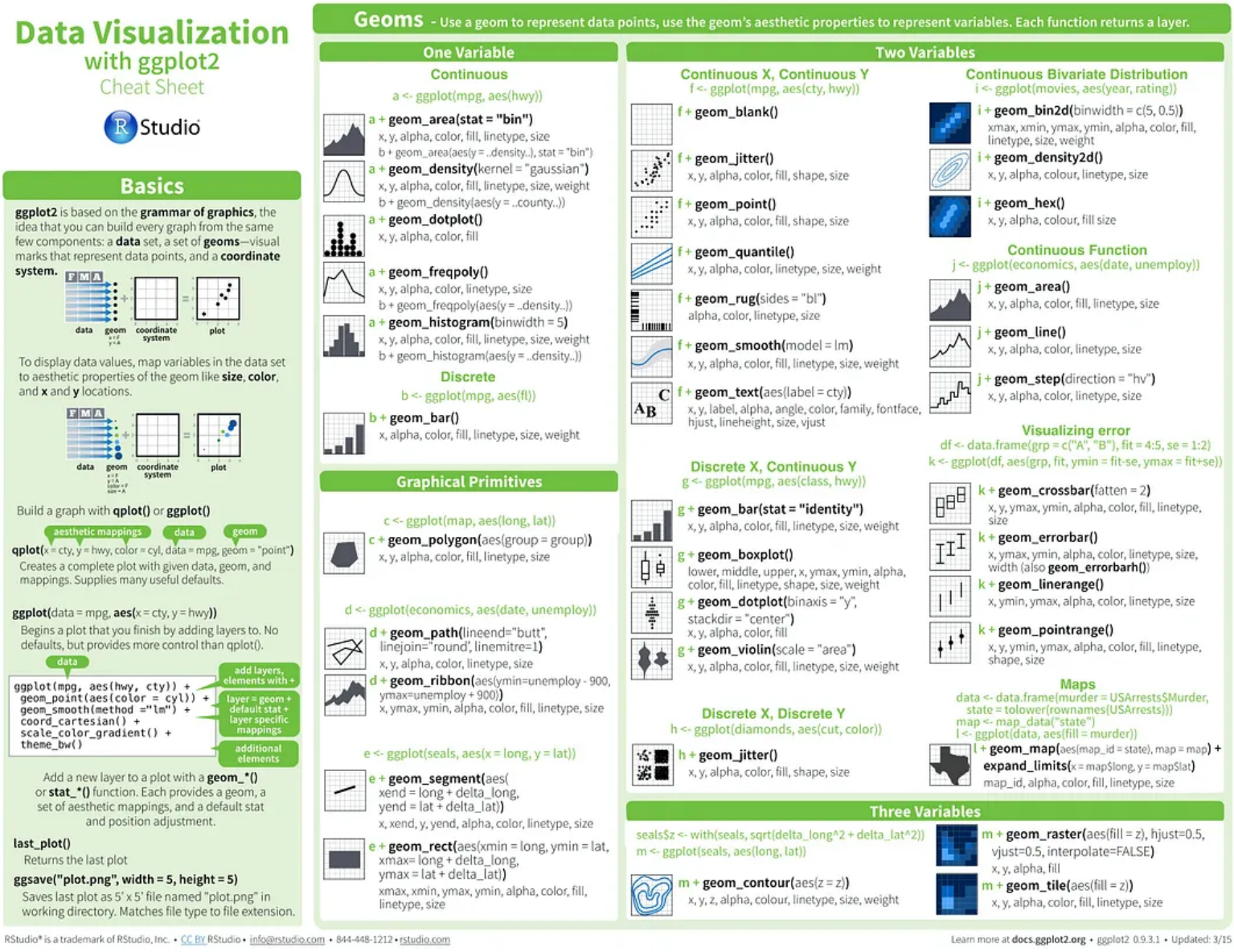


Neural Turing Machine (NTM)



Source — http://www.asimovinstitute.org/neural-network-zoo/

8. ggplot2



Data Visualization with ggplot2 Cheat Sheet

RStudio

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same few components: a **data** set, a set of **geoms**—visual marks that represent data points, and a **coordinate system.**

data · geom · coordinate system · plot

To display data values, map variables in the data set to aesthetic properties of the geom like **size**, **color**, and **x** and **y** locations.

data · geom · coordinate system · plot

Build a graph with **qplot()** or **ggplot()**

aesthetic mappings · data · geom

qplot(x = cty, y = hwy, color = cyl, data = mpg, geom = "point")
Creates a complete plot with given data, geom, and mappings. Supplies many useful defaults.

ggplot(data = mpg, **aes**(x = cty, y = hwy))
Begins a plot that you finish by adding layers to. No defaults, but provides more control than qplot().

data

```
ggplot(mpg, aes(hwy, cty)) +
 geom_point(aes(color = cyl)) +
 geom_smooth(method ="lm") +
 coord_cartesian() +
 scale_color_gradient() +
 theme_bw()
```

add layers, elements with +
layer = geom + default stat + layer specific mappings
additional elements

Add a new layer to a plot with a **geom_*()** or **stat_*()** function. Each provides a geom, a set of aesthetic mappings, and a default stat and position adjustment.

last_plot()
Returns the last plot

ggsave("plot.png", width = 5, height = 5)
Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Geoms - Use a geom to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

One Variable

Continuous

a <- ggplot(mpg, aes(hwy))

a + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size
b + geom_area(aes(y = ..density..), stat = "bin")

a + geom_density(kernel = "gaussian"**)**
x, y, alpha, color, fill, linetype, size, weight
b + geom_density(aes(y = ..county..))

a + geom_dotplot()
x, y, alpha, color, fill

a + geom_freqpoly()
x, y, alpha, color, linetype, size
b + geom_freqpoly(aes(y = ..density..))

a + geom_histogram(binwidth = 5**)**
x, y, alpha, color, fill, linetype, size, weight
b + geom_histogram(aes(y = ..density..))

Discrete

b <- ggplot(mpg, aes(fl))

b + geom_bar()
x, alpha, color, fill, linetype, size, weight

Graphical Primitives

c <- ggplot(map, aes(long, lat))

c + geom_polygon(aes(group = group)**)**
x, y, alpha, color, fill, linetype, size

d <- ggplot(economics, aes(date, unemploy))

d + geom_path(lineend="butt", linejoin="round", linemitre=1**)**
x, y, alpha, color, linetype, size

d + geom_ribbon(aes(ymin=unemploy - 900, ymax=unemploy + 900)**)**
x, ymax, ymin, alpha, color, fill, linetype, size

e <- ggplot(seals, aes(x = long, y = lat))

e + geom_segment(aes(xend = long + delta_long, yend = lat + delta_lat)**)**
x, xend, y, yend, alpha, color, linetype, size

e + geom_rect(aes(xmin = long, ymin = lat, xmax= long + delta_long, ymax = lat + delta_lat)**)**
xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

Two Variables

Continuous X, Continuous Y

f <- ggplot(mpg, aes(cty, hwy))

f + geom_blank()

f + geom_jitter()
x, y, alpha, color, fill, shape, size

f + geom_point()
x, y, alpha, color, fill, shape, size

f + geom_quantile()
x, y, alpha, color, linetype, size, weight

f + geom_rug(sides = "bl"**)**
alpha, color, linetype, size

f + geom_smooth(model = lm**)**
x, y, alpha, color, fill, linetype, size, weight

f + geom_text(aes(label = cty)**)**
x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

Discrete X, Continuous Y

g <- ggplot(mpg, aes(class, hwy))

g + geom_bar(stat = "identity")
x, y, alpha, color, fill, linetype, size, weight

g + geom_boxplot()
lower, middle, upper, x, ymax, ymin, alpha, color, fill, linetype, shape, size, weight

g + geom_dotplot(binaxis = "y", stackdir = "center"**)**
x, y, alpha, color, fill

g + geom_violin(scale = "area"**)**
x, y, alpha, color, fill, linetype, size, weight

Discrete X, Discrete Y

h <- ggplot(diamonds, aes(cut, color))

h + geom_jitter()
x, y, alpha, color, fill, shape, size

Continuous Bivariate Distribution

i <- ggplot(movies, aes(year, rating))

i + geom_bin2d(binwidth = c(5, 0.5)**)**
xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size, weight

i + geom_density2d()
x, y, alpha, colour, linetype, size

i + geom_hex()
x, y, alpha, colour, fill size

Continuous Function

j <- ggplot(economics, aes(date, unemploy))

j + geom_area()
x, y, alpha, color, fill, linetype, size

j + geom_line()
x, y, alpha, color, linetype, size

j + geom_step(direction = "hv"**)**
x, y, alpha, color, linetype, size

Visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
k <- ggplot(df, aes(grp, fit, ymin = fit-se, ymax = fit+se))

k + geom_crossbar(fatten = 2**)**
x, y, ymax, ymin, alpha, color, fill, linetype, size

k + geom_errorbar()
x, ymax, ymin, alpha, color, linetype, size, width (also **geom_errorbarh()**)

k + geom_linerange()
x, ymin, ymax, alpha, color, linetype, size

k + geom_pointrange()
x, y, ymin, ymax, alpha, color, fill, linetype, shape, size

Maps

data <- data.frame(murder = USArrests$Murder, state = tolower(rownames(USArrests)))
map <- map_data("state")
l <- ggplot(data, aes(fill = murder))

l + geom_map(aes(map_id = state), map = map**) +**
expand_limits(x = map$long, y = map$lat**)**
map_id, alpha, color, fill, linetype, size

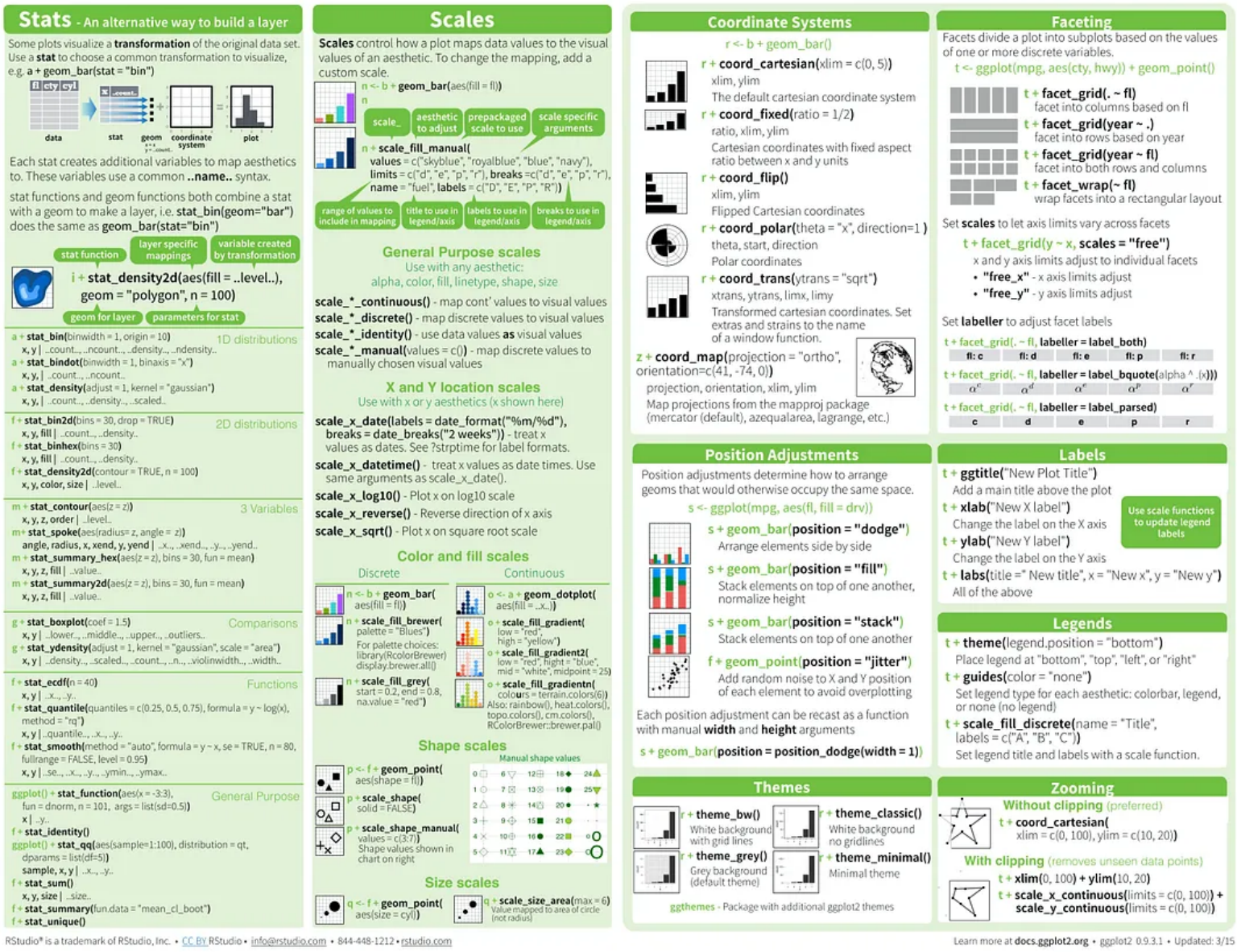
Three Variables

seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2))
m <- ggplot(seals, aes(long, lat))

m + geom_contour(aes(z = z)**)**
x, y, z, alpha, colour, linetype, size, weight

m + geom_raster(aes(fill = z), hjust=0.5, vjust=0.5, interpolate=FALSE**)**
x, y, alpha, fill

m + geom_tile(aes(fill = z)**)**
x, y, alpha, color, fill, linetype, size



Source — https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

9. PySpark

Python For Data Science *Cheat Sheet*

PySpark Basics

Learn Python for data science Interactively at www.DataCamp.com

Spark

PySpark is the Spark Python API that exposes the Spark programming model to Python

Initializing Spark

SparkContext

```
>>> from pyspark import SparkContext
>>> sc = SparkContext(master = 'local[2]')
```

Inspect SparkContext

```
>>> sc.version                  Retrieve SparkContext version
>>> sc.pythonVer                Retrieve Python version
>>> sc.master                   Master URL to connect to
>>> str(sc.sparkHome)           Path where Spark is installed on worker nodes
>>> str(sc.sparkUser())         Retrieve name of the Spark User running SparkContext
>>> sc.appName                  Return application name
>>> sc.applicationId            Retrieve application ID
>>> sc.defaultParallelism       Return default level of parallelism
>>> sc.defaultMinPartitions     Default minimum number of partitions for RDDs
```

Configuration

```
>>> from pyspark import SparkConf, SparkContext
>>> conf = (SparkConf()
            .setMaster("local")
            .setAppName("My app")
            .set("spark.executor.memory", "1g"))
>>> sc = SparkContext(conf = conf)
```

Using The Shell

In the PySpark shell, a special interpreter-aware SparkContext is already created in the variable called `sc`.

```
$ ./bin/spark-shell --master local[2]
$ ./bin/pyspark --master local[4] --py-files code.py
```

Set which master the context connects to with the `--master` argument, and add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to `--py-files`.

Loading Data

Parallelized Collections

```
>>> rdd = sc.parallelize([('a',7),('a',2),('b',2)])
>>> rdd2 = sc.parallelize([('a',2),('d',1),('b',1)])
>>> rdd3 = sc.parallelize(range(100))
>>> rdd4 = sc.parallelize([("a",["x","y","z"]),
                           ("b",["p", "r"])])
```

External Data

Read either one text file from HDFS, a local file system or or any Hadoop-supported file system URI with `textFile()`, or read in a directory of text files with `wholeTextFiles()`.

```
>>> textFile = sc.textFile("/my/directory/*.txt")
>>> textFile2 = sc.wholeTextFiles("/my/directory/")
```

Retrieving RDD Information

Basic Information

```
>>> rdd.getNumPartitions()      List the number of partitions
>>> rdd.count()                 Count RDD instances
 3
>>> rdd.countByKey()            Count RDD instances by key
 defaultdict(<type 'int'>,{'a':2,'b':1})
>>> rdd.countByValue()          Count RDD instances by value
 defaultdict(<type 'int'>,{('b',2):1,('a',2):1,('a',7):1})
>>> rdd.collectAsMap()          Return (key,value) pairs as a dictionary
 {'a': 2,'b': 2}
>>> rdd3.sum()                  Sum of RDD elements
 4950
>>> sc.parallelize([]).isEmpty() Check whether RDD is empty
 True
```

Summary

```
>>> rdd3.max()                  Maximum value of RDD elements
 99
>>> rdd3.min()                  Minimum value of RDD elements
 0
>>> rdd3.mean()                 Mean value of RDD elements
 49.5
>>> rdd3.stdev()                Standard deviation of RDD elements
 28.866070047722118
>>> rdd3.variance()             Compute variance of RDD elements
 833.25
>>> rdd3.histogram(3)           Compute histogram by bins
 ([0,33,66,99],[33,33,34])
>>> rdd3.stats()                Summary statistics (count, mean, stdev, max & min)
```

Applying Functions

```
>>> rdd.map(lambda x: x+(x[1],x[0])).collect()
 [('a',7,7,'a'),('a',2,2,'a'),('b',2,2,'b')]
>>> rdd5 = rdd.flatMap(lambda x: x+(x[1],x[0]))
>>> rdd5.collect()
 ['a',7,7,'a','a',2,2,'a','b',2,2,'b']
>>> rdd4.flatMapValues(lambda x: x).collect()
 [('a','x'),('a','y'),('a','z'),('b','p'),('b','r')]
```

Apply a function to each RDD element

Apply a function to each RDD element and flatten the result

Apply a flatMap function to each (key,value) pair of `rdd4` without changing the keys

Selecting Data

Getting

```
>>> rdd.collect()               Return a list with all RDD elements
 [('a', 7), ('a', 2), ('b', 2)]
>>> rdd.take(2)                 Take first 2 RDD elements
 [('a', 7), ('a', 2)]
>>> rdd.first()                 Take first RDD element
 ('a', 7)
>>> rdd.top(2)                  Take top 2 RDD elements
 [('b', 2), ('a', 7)]
```

Sampling

```
>>> rdd3.sample(False, 0.15, 81).collect()   Return sampled subset of rdd3
 [3,4,27,31,40,41,42,43,60,76,79,80,86,97]
```

Filtering

```
>>> rdd.filter(lambda x: "a" in x).collect()   Filter the RDD
 [('a',7),('a',2)]
>>> rdd5.distinct().collect()   Return distinct RDD values
 ['a',2,'b',7]
>>> rdd.keys().collect()        Return (key,value) RDD's keys
 ['a', 'a', 'b']
```

Iterating

```
>>> def g(x): print(x)
>>> rdd.foreach(g)              Apply a function to all RDD elements
 ('a', 7)
 ('b', 2)
 ('a', 2)
```

Reshaping Data

Reducing

```
>>> rdd.reduceByKey(lambda x,y : x+y)   Merge the rdd values for each key
       .collect()
 [('a',9),('b',2)]
>>> rdd.reduce(lambda a, b: a + b)      Merge the rdd values
 ('a',7,'a',2,'b',2)
```

Grouping by

```
>>> rdd3.groupBy(lambda x: x % 2)       Return RDD of grouped values
        .mapValues(list)
        .collect()
>>> rdd.groupByKey()()                  Group rdd by key
        .mapValues(list)
        .collect()
 [('a',[7,2]),('b',[2])]
```

Aggregating

```
>>> seqOp = (lambda x,y: (x[0]+y,x[1]+1))
>>> combOp = (lambda x,y:(x[0]+y[0],x[1]+y[1]))
>>> rdd3.aggregate((0,0),seqOp,combOp)  Aggregate RDD elements of each partition and then the results
 (4950,100)
>>> rdd.aggregateByKey((0,0),seqop,combop)   Aggregate values of each RDD key
       .collect()
 [('a',(9,2)), ('b',(2,1))]
>>> rdd3.fold(0,add)                    Aggregate the elements of each partition, and then the results
 4950
>>> rdd.foldByKey(0, add)               Merge the values for each key
       .collect()
 [('a',9),('b',2)]
>>> rdd3.keyBy(lambda x: x+x)           Create tuples of RDD elements by applying a function
       .collect()
```

Mathematical Operations

```
>>> rdd.subtract(rdd2)                  Return each rdd value not contained in rdd2
       .collect()
 [('b',2),('a',7)]
>>> rdd2.subtractByKey(rdd)             Return each (key,value) pair of rdd2 with no matching key in rdd
       .collect()
 [('d', 1)]
>>> rdd.cartesian(rdd2).collect()       Return the Cartesian product of rdd and rdd2
```

Sort

```
>>> rdd2.sortBy(lambda x: x[1])         Sort RDD by given function
       .collect()
 [('d',1),('b',1),('a',2)]
>>> rdd2.sortByKey()()                  Sort (key, value) RDD by key
       .collect()
 [('a',2),('b',1),('d',1)]
```

Repartitioning

```
>>> rdd.repartition(4)          New RDD with 4 partitions
>>> rdd.coalesce(1)             Decrease the number of partitions in the RDD to 1
```

Saving

```
>>> rdd.saveAsTextFile("rdd.txt")
>>> rdd.saveAsHadoopFile("hdfs://namenodehost/parent/child",
                         'org.apache.hadoop.mapred.TextOutputFormat')
```

Stopping SparkContext

```
>>> sc.stop()
```

Execution

```
$ ./bin/spark-submit examples/src/main/python/pi.py
```

DataCamp
Learn Python for Data Science Interactively

Source — https://www.datacamp.com/community/blog/pyspark-cheat-sheet-python#gs.L=J1zxQ

Python For Data Science *Cheat Sheet*

PySpark - RDD Basics

Learn Python for data science Interactively at www.DataCamp.com

Spark

PySpark is the Spark Python API that exposes the Spark programming model to Python.

Initializing Spark

SparkContext

```
>>> from pyspark import SparkContext
>>> sc = SparkContext(master = 'local[2]')
```

Inspect SparkContext

```
>>> sc.version                  Retrieve SparkContext version
>>> sc.pythonVer                Retrieve Python version
>>> sc.master                   Master URL to connect to
>>> str(sc.sparkHome)           Path where Spark is installed on worker nodes
>>> str(sc.sparkUser())         Retrieve name of the Spark User running SparkContext
>>> sc.appName                  Return application name
>>> sc.applicationId            Retrieve application ID
>>> sc.defaultParallelism       Return default level of parallelism
>>> sc.defaultMinPartitions     Default minimum number of partitions for RDDs
```

Configuration

```
>>> from pyspark import SparkConf, SparkContext
>>> conf = (SparkConf()
            .setMaster("local")
            .setAppName("My app")
            .set("spark.executor.memory", "1g"))
>>> sc = SparkContext(conf = conf)
```

Using The Shell

In the PySpark shell, a special interpreter-aware SparkContext is already created in the variable called `sc`.

```
$ ./bin/spark-shell --master local[2]
$ ./bin/pyspark --master local[4] --py-files code.py
```

Set which master the context connects to with the `--master` argument, and add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to `--py-files`.

Loading Data

Parallelized Collections

```
>>> rdd = sc.parallelize([('a',7),('a',2),('b',2)])
>>> rdd2 = sc.parallelize([('a',2),('d',1),('b',1)])
>>> rdd3 = sc.parallelize(range(100))
>>> rdd4 = sc.parallelize([("a",["x","y","z"]),
                           ("b",["p", "r"])])
```

External Data

Read either one text file from HDFS, a local file system or or any Hadoop-supported file system URI with `textFile()`, or read in a directory of text files with `wholeTextFiles()`.

```
>>> textFile = sc.textFile("/my/directory/*.txt")
>>> textFile2 = sc.wholeTextFiles("/my/directory/")
```

Retrieving RDD Information

Basic Information

```
>>> rdd.getNumPartitions()          List the number of partitions
>>> rdd.count()                     Count RDD instances
3
>>> rdd.countByKey()                Count RDD instances by key
defaultdict(<type 'int'>,{'a':2,'b':1})
>>> rdd.countByValue()              Count RDD instances by value
defaultdict(<type 'int'>,{('b',2):1,('a',2):1,('a',7):1})
>>> rdd.collectAsMap()              Return (key,value) pairs as a dictionary
{'a': 2,'b': 2}
>>> rdd3.sum()                      Sum of RDD elements
4950
>>> sc.parallelize([]).isEmpty()    Check whether RDD is empty
True
```

Summary

```
>>> rdd3.max()                      Maximum value of RDD elements
99
>>> rdd3.min()                      Minimum value of RDD elements
0
>>> rdd3.mean()                     Mean value of RDD elements
49.5
>>> rdd3.stdev()                    Standard deviation of RDD elements
28.866070047722118
>>> rdd3.variance()                 Compute variance of RDD elements
833.25
>>> rdd3.histogram(3)               Compute histogram by bins
([0,33,66,99],[33,33,34])
>>> rdd3.stats()                    Summary statistics (count, mean, stdev, max & min)
```

Applying Functions

```
>>> rdd.map(lambda x: x+(x[1],x[0]))          Apply a function to each RDD element
      .collect()
  [('a',7,7,'a'),('a',2,2,'a'),('b',2,2,'b')]
>>> rdd5 = rdd.flatMap(lambda x: x+(x[1],x[0]))   Apply a function to each RDD element and flatten the result

>>> rdd5.collect()
  ['a',7,7,'a','a',2,2,'a','b',2,2,'b']
>>> rdd4.flatMapValues(lambda x: x)           Apply a flatMap function to each (key,value) pair of rdd4 without changing the keys
      .collect()
  [('a','x'),('a','y'),('a','z'),('b','p'),('b','r')]
```

Selecting Data

Getting

```
>>> rdd.collect()                   Return a list with all RDD elements
  [('a', 7), ('a', 2), ('b', 2)]
>>> rdd.take(2)                     Take first 2 RDD elements
  [('a', 7), ('a', 2)]
>>> rdd.first()                     Take first RDD element
  ('a', 7)
>>> rdd.top(2)                      Take top 2 RDD elements
  [('b', 2), ('a', 7)]
```

Sampling

```
>>> rdd3.sample(False, 0.15, 81).collect()   Return sampled subset of rdd3
  [3,4,27,31,40,41,42,43,60,76,79,80,86,97]
```

Filtering

```
>>> rdd.filter(lambda x: "a" in x)  Filter the RDD
      .collect()
  [('a',7),('a',2)]
>>> rdd5.distinct().collect()       Return distinct RDD values
  ['a',2,'b',7]
>>> rdd.keys().collect()            Return (key,value) RDD's keys
  ['a', 'a', 'b']
```

Iterating

```
>>> def g(x): print(x)
>>> rdd.foreach(g)                  Apply a function to all RDD elements
  ('a', 7)
  ('b', 2)
  ('a', 2)
```

Reshaping Data

Reducing

```
>>> rdd.reduceByKey(lambda x,y : x+y)   Merge the rdd values for each key
      .collect()
  [('a',9),('b',2)]
>>> rdd.reduce(lambda a, b: a + b)      Merge the rdd values
  ('a',7,'a',2,'b',2)
```

Grouping by

```
>>> rdd3.groupBy(lambda x: x % 2)       Return RDD of grouped values
      .mapValues(list)
      .collect()
>>> rdd.groupByKey()                    Group rdd by key
      .mapValues(list)
      .collect()
  [('a',[7,2]),('b',[2])]
```

Aggregating

```
>>> seqOp = (lambda x,y: (x[0]+y,x[1]+1))
>>> combOp = (lambda x,y:(x[0]+y[0],x[1]+y[1]))
>>> rdd3.aggregate((0,0),seqOp,combOp)  Aggregate RDD elements of each partition and then the results
  (4950,100)
>>> rdd.aggregateByKey((0,0),seqop,combop)   Aggregate values of each RDD key
      .collect()
  [('a',(9,2)), ('b',(2,1))]
>>> rdd3.fold(0,add)                    Aggregate the elements of each partition, and then the results
  4950
>>> rdd.foldByKey(0, add)               Merge the values for each key
      .collect()
  [('a',9),('b',2)]
>>> rdd3.keyBy(lambda x: x+x)           Create tuples of RDD elements by applying a function
      .collect()
```

Mathematical Operations

```
>>> rdd.subtract(rdd2)                  Return each rdd value not contained in rdd2
      .collect()
  [('b',2),('a',7)]
>>> rdd2.subtractByKey(rdd)             Return each (key,value) pair of rdd2 with no matching key in rdd
      .collect()
  [('d', 1)]
>>> rdd.cartesian(rdd2).collect()       Return the Cartesian product of rdd and rdd2
```

Sort

```
>>> rdd2.sortBy(lambda x: x[1])         Sort RDD by given function
      .collect()
  [('d',1),('b',1),('a',2)]
>>> rdd2.sortByKey()                    Sort (key, value) RDD by key
      .collect()
  [('a',2),('b',1),('d',1)]
```

Repartitioning

```
>>> rdd.repartition(4)                  New RDD with 4 partitions
>>> rdd.coalesce(1)                     Decrease the number of partitions in the RDD to 1
```

Saving

```
>>> rdd.saveAsTextFile("rdd.txt")
>>> rdd.saveAsHadoopFile("hdfs://namenodehost/parent/child",
                         'org.apache.hadoop.mapred.TextOutputFormat')
```

Stopping SparkContext

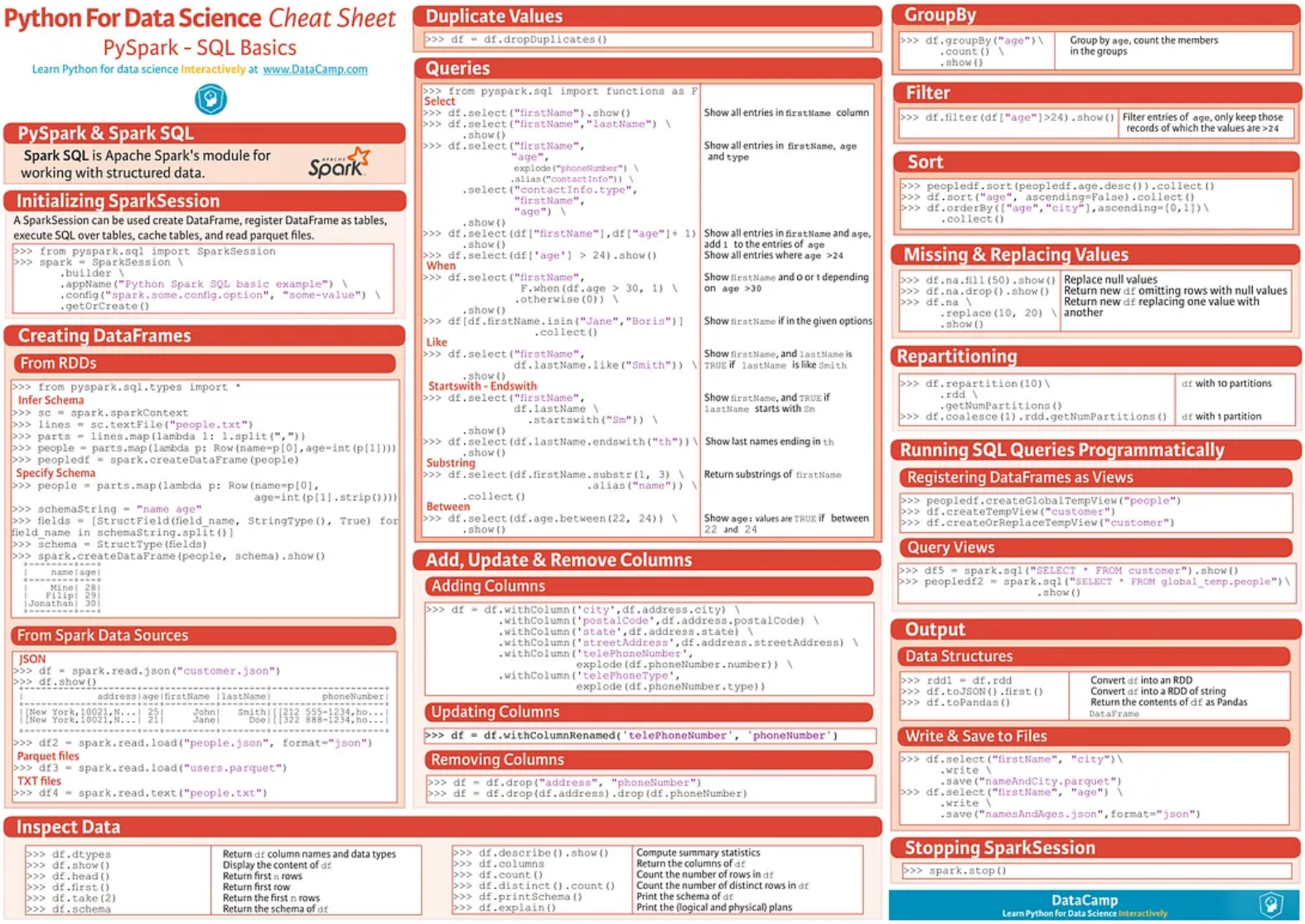
```
>>> sc.stop()
```

Execution

```
$ ./bin/spark-submit examples/src/main/python/pi.py
```

DataCamp
Learn Python for Data Science Interactively

Source — https://www.datacamp.com/community/blog/pyspark-cheat-sheet-python#gs.L=J1zxQ



Source — https://www.datacamp.com/community/blog/pyspark-sql-cheat-sheet

10. R Studio (dplyr and tidyr)

Data Wrangling with dplyr and tidyr Cheat Sheet

RStudio

Syntax - Helpful conventions for wrangling

dplyr::tbl_df(iris)

Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

```
Source: local data frame [150 x 5]

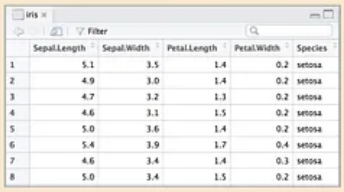
   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: Petal.Width (dbl),
  Species (fctr)
```

dplyr::glimpse(iris)

Information dense summary of tbl data.

utils::View(iris)

View data set in spreadsheet-like display (note capital V).



dplyr::%>%

Passes object on left hand side as first argument (or . argument) of function on righthand side.

```
x %>% f(y)  is the same as  f(x, y)
y %>% f(x, .., z)  is the same as  f(x, y, z )
```

"Piping" with %>% makes code more readable, e.g.

```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

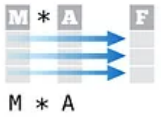
Tidy Data - A foundation for wrangling in R

In a tidy data set:

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.



Reshaping Data - Change the layout of a data set

tidyr::gather(cases, "year", "n", 2:4)

Gather columns into rows.

tidyr::spread(pollution, size, amount)

Spread rows into columns.

tidyr::separate(storms, date, c("y", "m", "d"))

Separate one column into several.

tidyr::unite(data, col, ..., sep)

Unite several columns into one.

dplyr::data_frame(a = 1:3, b = 4:6)

Combine vectors into data frame (optimized).

dplyr::arrange(mtcars, mpg)

Order rows by values of a column (low to high).

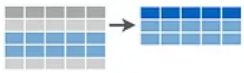
dplyr::arrange(mtcars, desc(mpg))

Order rows by values of a column (high to low).

dplyr::rename(tb, y = year)

Rename the columns of a data frame.

Subset Observations (Rows)



dplyr::filter(iris, Sepal.Length > 7)

Extract rows that meet logical criteria.

dplyr::distinct(iris)

Remove duplicate rows.

dplyr::sample_frac(iris, 0.5, replace = TRUE)

Randomly select fraction of rows.

dplyr::sample_n(iris, 10, replace = TRUE)

Randomly select n rows.

dplyr::slice(iris, 10:15)

Select rows by position.

dplyr::top_n(storms, 2, date)

Select and order top n entries (by group if grouped data).

Logic in R - ?Comparison, ?base::Logic

<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&, |, !, xor, any, all	Boolean operators

Subset Variables (Columns)

dplyr::select(iris, Sepal.Width, Petal.Length, Species)

Select columns by name or helper function.

Helper functions for select - ?select

select(iris, contains("."))
Select columns whose name contains a character string.

select(iris, ends_with("Length"))
Select columns whose name ends with a character string.

select(iris, everything())
Select every column.

select(iris, matches(".t."))
Select columns whose name matches a regular expression.

select(iris, num_range("x", 1:5))
Select columns named x1, x2, x3, x4, x5.

select(iris, one_of(c("Species", "Genus")))
Select columns whose names are in a group of names.

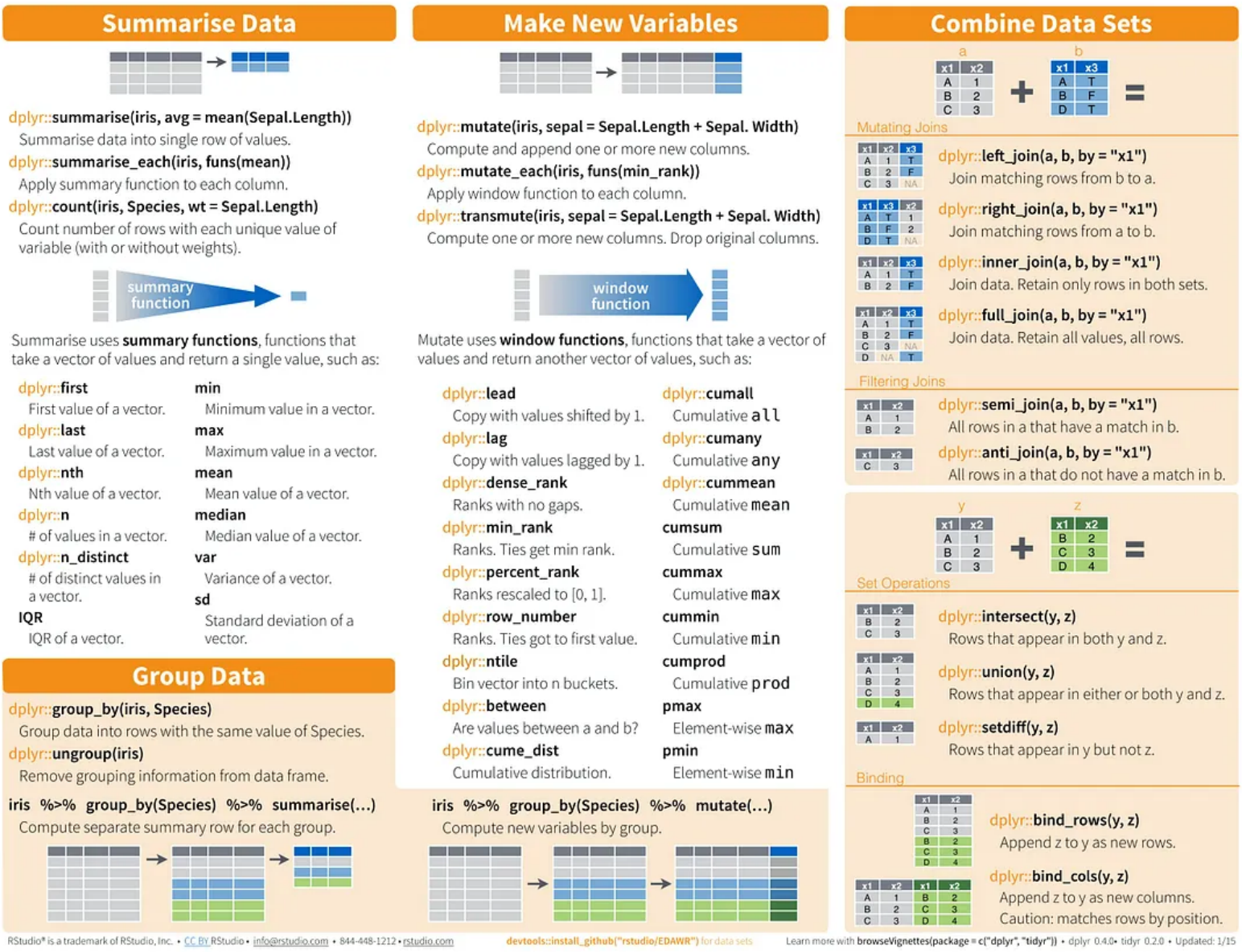
select(iris, starts_with("Sepal"))
Select columns whose name starts with a character string.

select(iris, Sepal.Length:Petal.Width)
Select all columns between Sepal.Length and Petal.Width (inclusive).

select(iris, -Species)
Select all columns except Species.

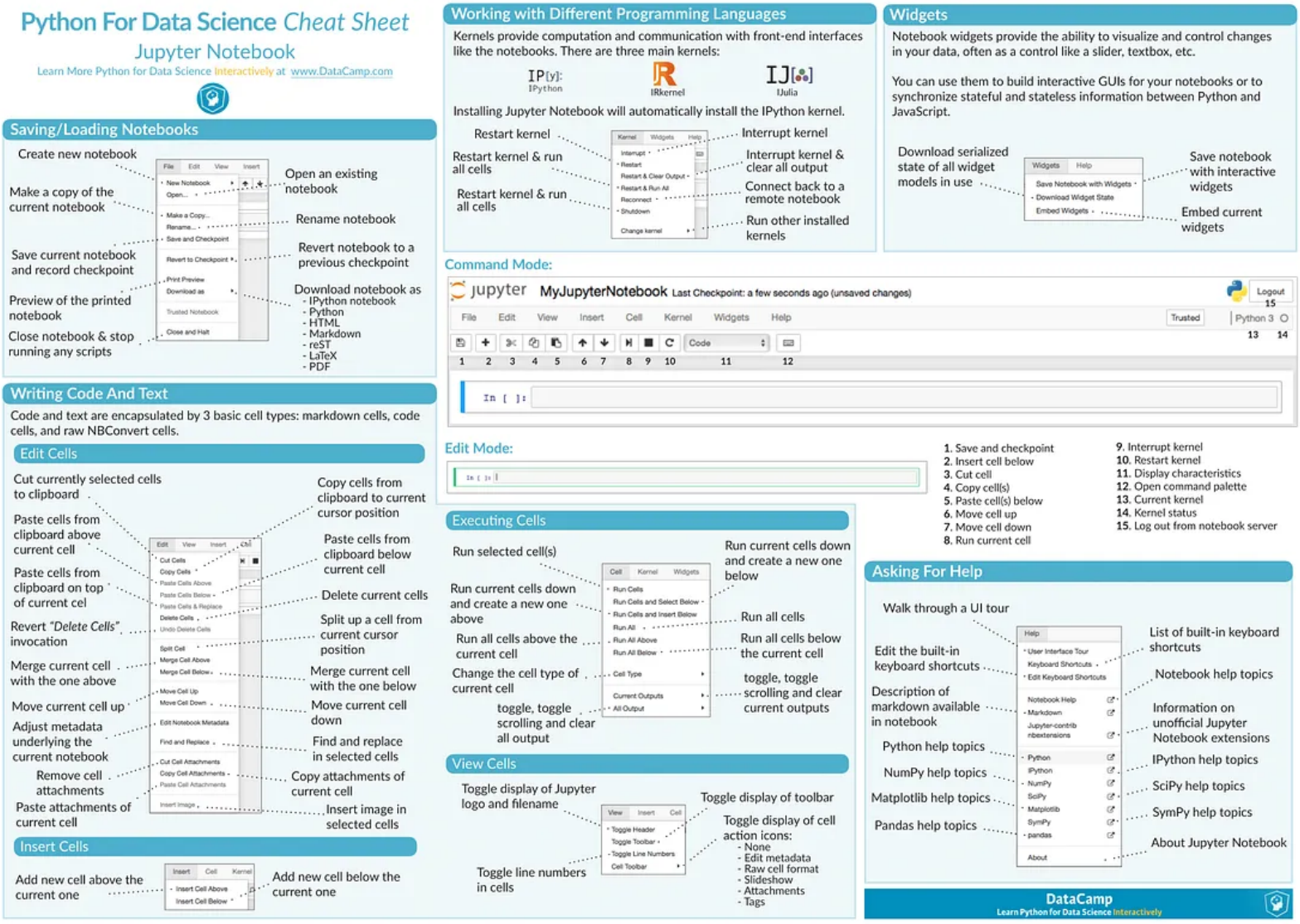
devtools::install_github("rstudio/EDAWR") for data sets

Learn more with browseVignettes(package = c("dplyr", "tidyr")) • dplyr 0.4.0 • tidyr 0.2.0 • Updated: 1/15



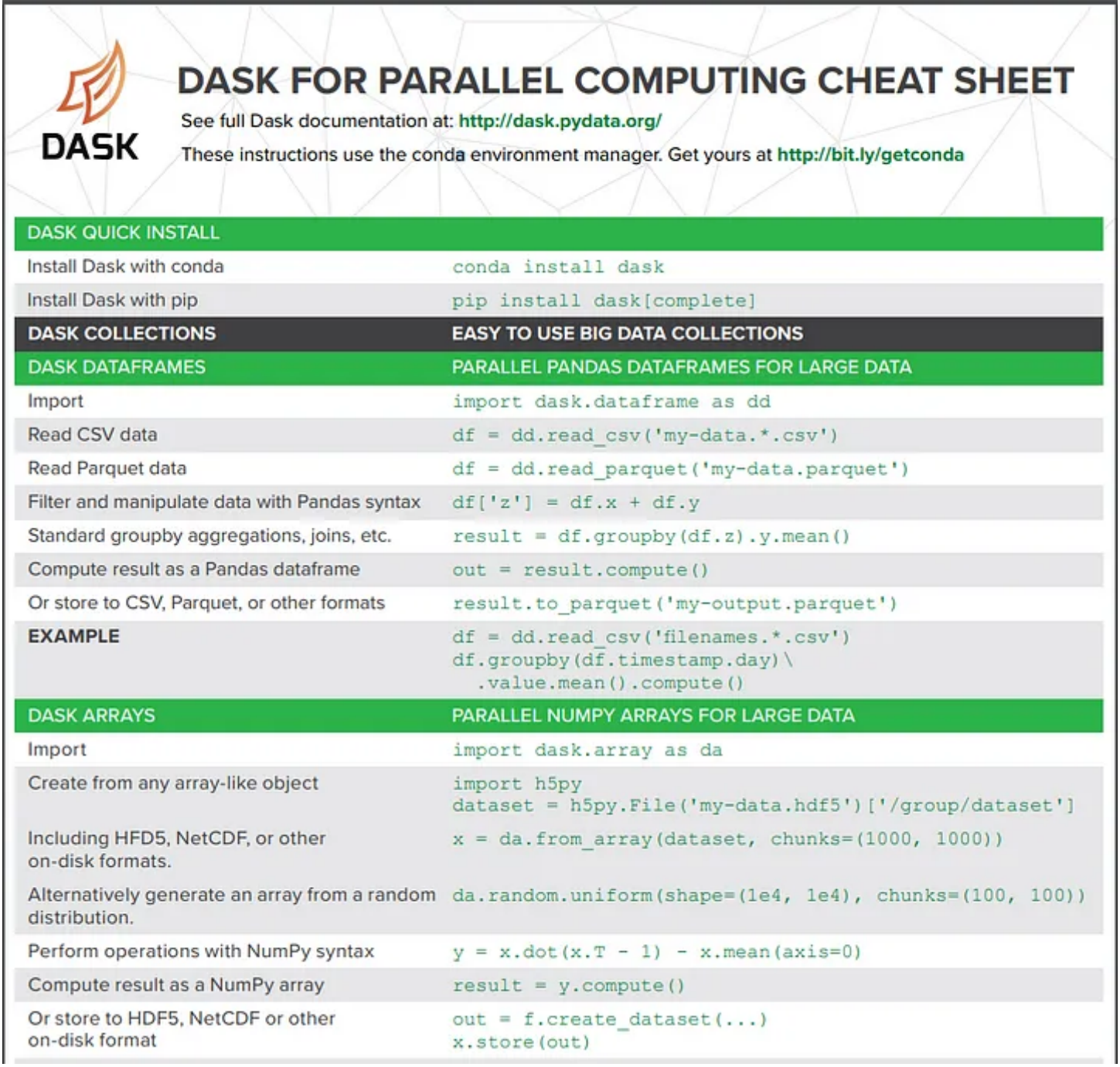
Source — https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

11. Jupyter Notebook



Source — https://www.datacamp.com/community/blog/jupyter-notebook-cheat-sheet

12. Dask



Source — http://docs.dask.org/en/latest/_downloads/daskcheatsheet.pdf

EXAMPLE	`with h5py.File('my-data.hdf5') as f:`
	`  x = da.from_array(f['/path'], chunks=(1000, 1000))`
	`  x -= x.mean(axis=0)`
	`  out = f.create_dataset(...)`
	`  x.store(out)`

DASK BAGS	**PARELLEL LISTS FOR UNSTRUCTURED DATA**
Import	`import dask.bag as db`
Create Dask Bag from a sequence	`b = db.from_sequence(seq, npartitions)`
Or read from text formats	`b = db.read_text('my-data.*.json')`

Map and filter results	`import json`
	`records = b.map(json.loads)`
	`  .filter(lambda d: d["name"] == "Alice")`

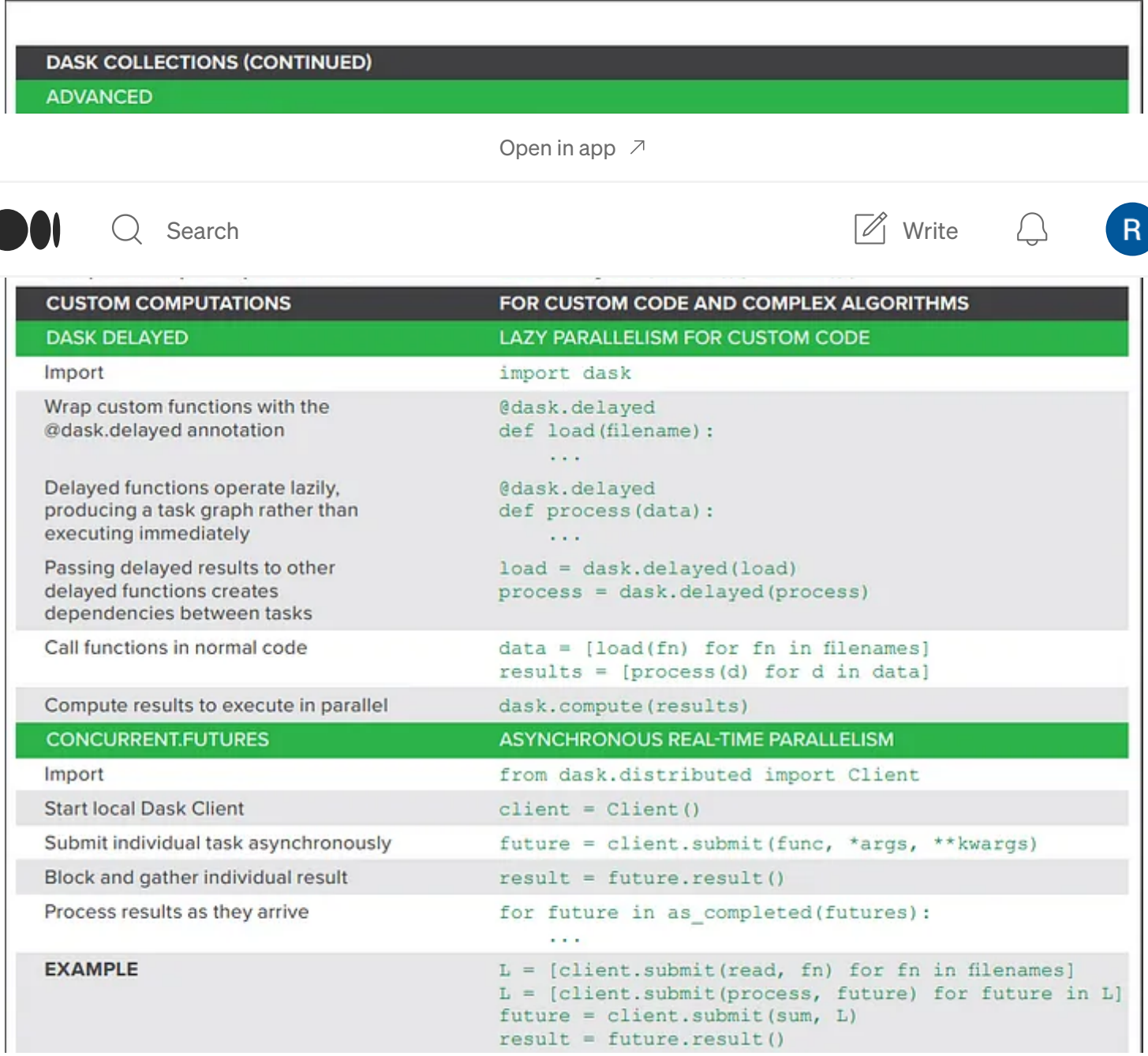
Compute aggregations like mean, count, sum	`records.pluck('key-name').mean().compute()`
Or store results back to text formats	`records.to_textfiles('output.*.json')`

EXAMPLE	`db.read_text('s3://bucket/my-data.*.json')`
	`  .map(json.loads)`
	`  .filter(lambda d: d["name"] == "Alice")`
	`  .to_textfiles('s3://bucket/output.*.json')`

ANACONDA.

CONTINUED ON BACK →

Source — http://docs.dask.org/en/latest/_downloads/daskcheatsheet.pdf



DASK COLLECTIONS (CONTINUED)	
ADVANCED	
CUSTOM COMPUTATIONS	**FOR CUSTOM CODE AND COMPLEX ALGORITHMS**
DASK DELAYED	LAZY PARALLELISM FOR CUSTOM CODE
Import	`import dask`

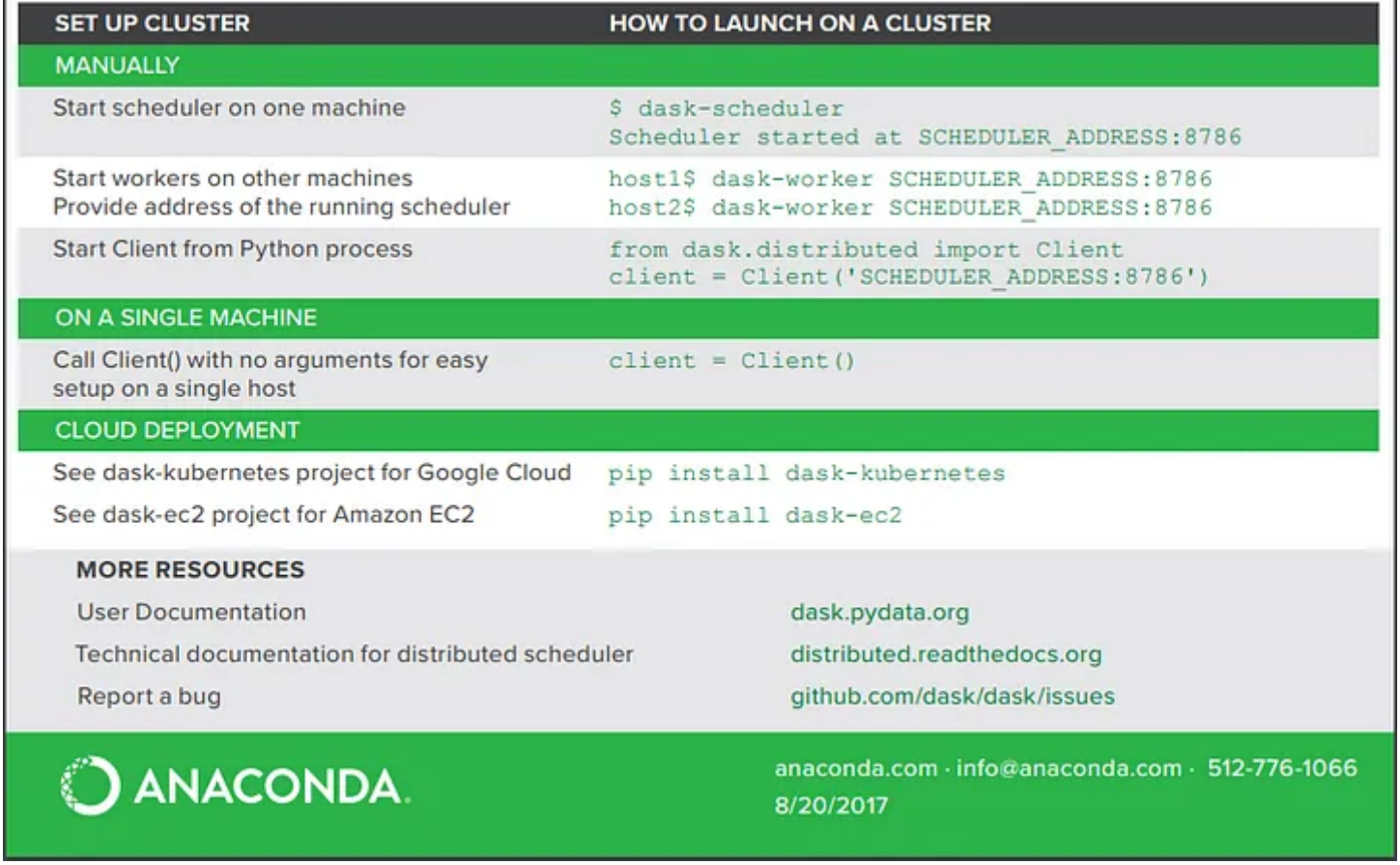
Wrap custom functions with the @dask.delayed annotation	`@dask.delayed`
	`def load(filename):`
	`  ...`

Delayed functions operate lazily, producing a task graph rather than executing immediately	`@dask.delayed`
	`def process(data):`
	`  ...`

Passing delayed results to other delayed functions creates dependencies between tasks	`load = dask.delayed(load)` `process = dask.delayed(process)`
Call functions in normal code	`data = [load(fn) for fn in filenames]` `results = [process(d) for d in data]`
Compute results to execute in parallel	`dask.compute(results)`
CONCURRENT.FUTURES	ASYNCHRONOUS REAL-TIME PARALLELISM
Import	`from dask.distributed import Client`
Start local Dask Client	`client = Client()`
Submit individual task asynchronously	`future = client.submit(func, *args, **kwargs)`
Block and gather individual result	`result = future.result()`
Process results as they arrive	`for future in as_completed(futures):` `  ...`

EXAMPLE	`L = [client.submit(read, fn) for fn in filenames]`
	`L = [client.submit(process, future) for future in L]`
	`future = client.submit(sum, L)`
	`result = future.result()`

Source — http://docs.dask.org/en/latest/_downloads/daskcheatsheet.pdf



SET UP CLUSTER	HOW TO LAUNCH ON A CLUSTER
MANUALLY	
Start scheduler on one machine	`$ dask-scheduler` `Scheduler started at SCHEDULER_ADDRESS:8786`
Start workers on other machines Provide address of the running scheduler	`host1$ dask-worker SCHEDULER_ADDRESS:8786` `host2$ dask-worker SCHEDULER_ADDRESS:8786`
Start Client from Python process	`from dask.distributed import Client` `client = Client('SCHEDULER_ADDRESS:8786')`
ON A SINGLE MACHINE	
Call Client() with no arguments for easy setup on a single host	`client = Client()`
CLOUD DEPLOYMENT	
See dask-kubernetes project for Google Cloud	`pip install dask-kubernetes`
See dask-ec2 project for Amazon EC2	`pip install dask-ec2`

Source — http://docs.dask.org/en/latest/_downloads/daskcheatsheet.pdf

> *Thank you for reading.*
>
> *If you want to get into contact, you can reach out to me at* ***ahikailash1@gmail.com***

About Me:

I am a Co-Founder of MateLabs, where we have built Mateverse, an ML Platform which enables everyone to easily build and train Machine Learning Models, without writing a single line of code.

> ***Note****: Recently, I published a book on GANs titled "Generative Adversarial Networks Projects", in which I covered most of the widely popular GAN architectures and their implementations. DCGAN, StackGAN, CycleGAN, Pix2pix, Age-cGAN, and 3D-GAN have been covered in details at the implementation level. Each architecture has a chapter dedicated to it. I have explained these networks in a very simple and descriptive language using Keras framework with Tensorflow*

backend. If you are working on GANs or planning to use GANs, give it a read and share your valuable feedback with me at ***ahikailash1@gmail.com***

Generative Adversarial Networks Projects: Build next-generation generative models using TensorFlow...

Explore various Generative Adversarial Network architectures using the Python ecosystem Key Features Use different...

www.amazon.com

You can grab a copy of the book from http://www.amazon.com/Generative-Adversarial-Networks-Projects-next-generation/dp/1789136679https://www.amazon.in/Generative-Adversarial-Networks-Projects-next-generation/dp/1789136679?fbclid=IwAR0X2pDk4CTxn5GqWmBbKIgiB38WmFX-sqCpBNI8k9Z8I-KCQ7VWRpJXm7I https://www.packtpub.com/big-data-and-business-intelligence/generative-adversarial-networks-projects?fbclid=IwAR2OtU21faMFPM4suH_HJmy_DRQxOVwJZB0kz3ZiSbFb_MW7INYCqqV7U0c

Machine Learning | Artificial Intelligence | Deep Learning | Technology | Computer Science

Written by Kailash Ahirwar

Follow

5.8K Followers · Writer for Startups & Venture Capital

Decentralizing Artificial Intelligence | Co-founder - Raven Protocol | Mate Labs | Author - Generative Adversarial Networks Projects

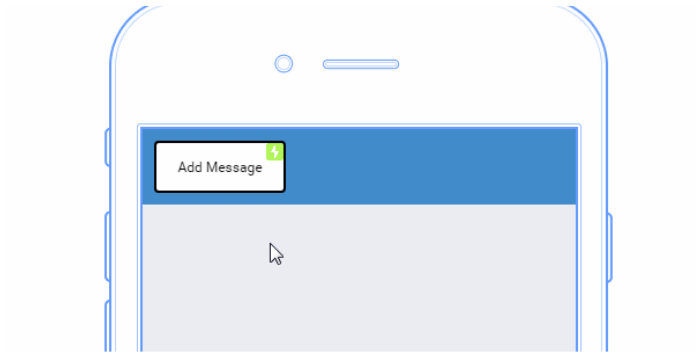
More from Kailash Ahirwar and Startups & Venture Capital

Kailash Ahirwar

A Very Short Introduction to Diffusion Models

Artificial Intelligence is constantly evolving to solve hard and complex problems. Image...

4 min read · Sep 26



Sophie Paxton in Startups & Venture Capital

Your UI isn't a Disney Movie

My previous short article about gratuitous animation really struck a chord with people....

5 min read · Oct 6, 2015

129 

Charles R Poliquin in Startups & Venture Capital

10 Household Items That Are Dropping Your Testosterone Level...

We live in a toxic world, and this toxicity affects androgen levels at an epidemic level

7 min read · Jan 19, 2018

879 1

4.5K 113

Kailash Ahirwar

A Very Short Introduction to Inception Score(IS)

Generative Adversarial Networks or GANs for short were successful in generating high-...

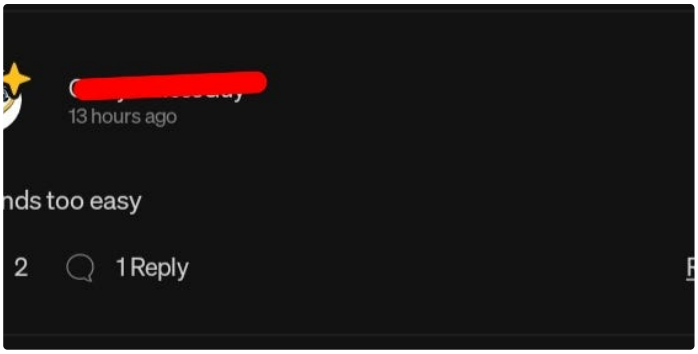
4 min read · Feb 24, 2021

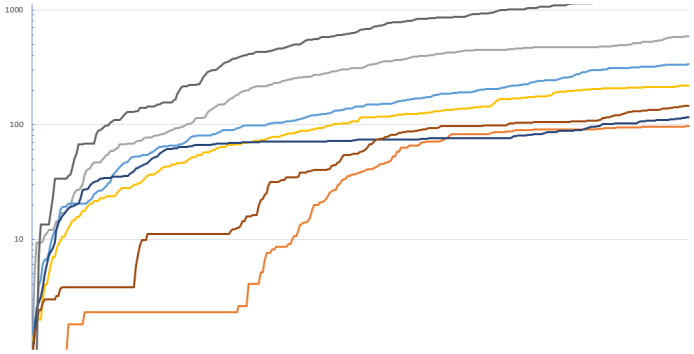
85

See all from Kailash Ahirwar

See all from Startups & Venture Capital

Recommended from Medium





Paul Rose

I Found A Very Profitable AI Side Hustle

And it's perfect for beginners

6 min read · Oct 19

12.7K 222

Pau Blasco i Roca in Towards Data Science

My Life Stats: I Tracked My Habits for a Year, and This Is What I...

I measured the time I spent on my daily activities (studying, doing sports, socializing...

12 min read · Nov 21

3.1K 57

Lists

Predictive Modeling w/ Python

20 stories · 659 saves

AI Regulation

6 stories · 208 saves

Natural Language Processing

924 stories · 441 saves

Practical Guides to Machine Learning

10 stories · 739 saves

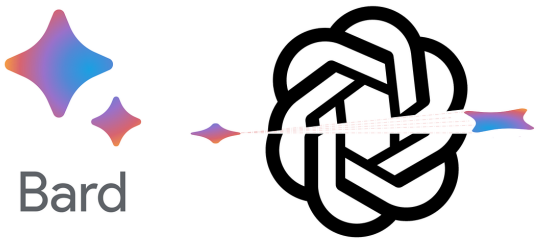
Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

Practical tips for those who want to advance in their careers

22 min read · Mar 21

12.4K 238



AL Anany

The ChatGPT Hype Is Over—Now Watch How Google Will Kill...

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 1

20K 615

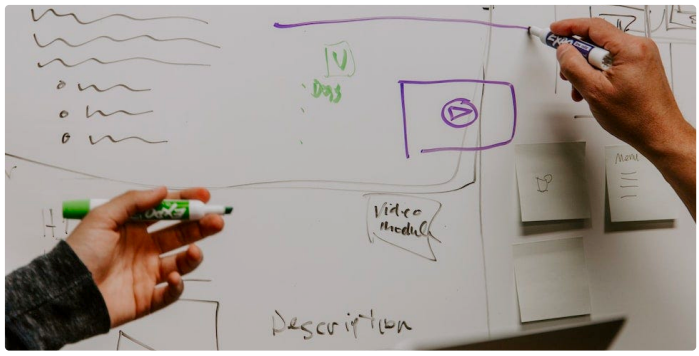
Data Scian by Imad Adrees

Best Portfolio Projects for Data Science

“How can I showcase my data skills to the world?” you may be asking. Fear not, for the...

5 min read · Sep 19

611 5



Carlos Arguelles

My favorite coding question to give candidates

A coding question, from the viewpoint of an Google/Amazon/Microsoft interviewer

11 min read · Nov 12

4.5K 49

See more recommendations

Python For Data Science *Cheat Sheet*

Python Basics

Learn More Python for Data Science Interactively at www.datacamp.com

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
 5
```

Calculations With Variables

`>>> x+2` → `7`	Sum of two variables
`>>> x-2` → `3`	Subtraction of two variables
`>>> x*2` → `10`	Multiplication of two variables
`>>> x**2` → `25`	Exponentiation of a variable
`>>> x%2` → `1`	Remainder of a variable
`>>> x/float(2)` → `2.5`	Division of a variable

Types and Type Conversion

`str()`	`'5', '3.45', 'True'`	Variables to strings
`int()`	`5, 3, 1`	Variables to integers
`float()`	`5.0, 1.0`	Variables to floats
`bool()`	`True, True, True`	Variables to booleans

Asking For Help

```
>>> help(str)
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset	
`>>> my_list[1]`	Select item at index 1
`>>> my_list[-3]`	Select 3rd last item
Slice	
`>>> my_list[1:3]`	Select items at index 1 and 2
`>>> my_list[1:]`	Select items after index 0
`>>> my_list[:3]`	Select items before index 3
`>>> my_list[:]`	Copy my_list
Subset Lists of Lists	
`>>> my_list2[1][0]`	my_list[list][itemOfList]
`>>> my_list2[1][:2]`	

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

`>>> my_list.index(a)`	Get the index of an item
`>>> my_list.count(a)`	Count an item
`>>> my_list.append('!')`	Append an item at a time
`>>> my_list.remove('!')`	Remove an item
`>>> del(my_list[0:1])`	Remove an item
`>>> my_list.reverse()`	Reverse the list
`>>> my_list.extend('!')`	Append an item
`>>> my_list.pop(-1)`	Remove an item
`>>> my_list.insert(0,'!')`	Insert an item
`>>> my_list.sort()`	Sort the list

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

pandas — Data analysis

learn — Machine learning

NumPy — Scientific computing

matplotlib — 2D plotting

Install Python

ANACONDA — Leading open data science platform powered by Python

spyder — Free IDE that is included with Anaconda

jupyter — Create and share documents with live code, visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset	
`>>> my_array[1]` → `2`	Select item at index 1
Slice	
`>>> my_array[0:2]` → `array([1, 2])`	Select items at index 0 and 1
Subset 2D Numpy arrays	
`>>> my_2darray[:,0]` → `array([1, 4])`	my_2darray[rows, columns]

Numpy Array Operations

```
>>> my_array > 3
array([False, False, False,  True], dtype=bool)
>>> my_array * 2
array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
array([6, 8, 10, 12])
```

Numpy Array Functions

`>>> my_array.shape`	Get the dimensions of the array
`>>> np.append(other_array)`	Append items to an array
`>>> np.insert(my_array, 1, 5)`	Insert items in an array
`>>> np.delete(my_array,[1])`	Delete items in an array
`>>> np.mean(my_array)`	Mean of the array
`>>> np.median(my_array)`	Median of the array
`>>> my_array.corrcoef()`	Correlation coefficient
`>>> np.std(my_array)`	Standard deviation

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

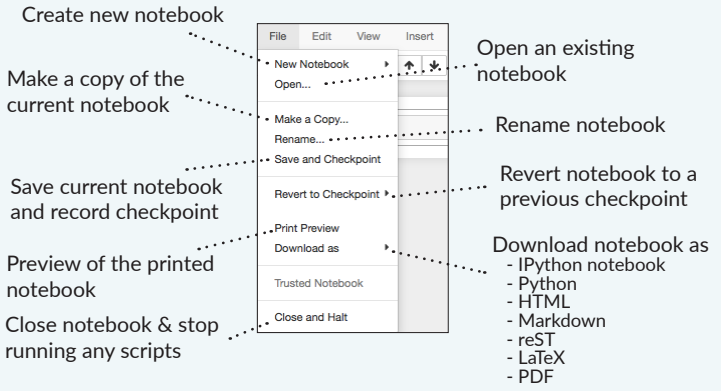
`>>> my_string.upper()`	String to uppercase
`>>> my_string.lower()`	String to lowercase
`>>> my_string.count('w')`	Count String elements
`>>> my_string.replace('e', 'i')`	Replace String elements
`>>> my_string.strip()`	Strip whitespaces

Python For Data Science *Cheat Sheet*

Jupyter Notebook

Learn More Python for Data Science Interactively at www.DataCamp.com

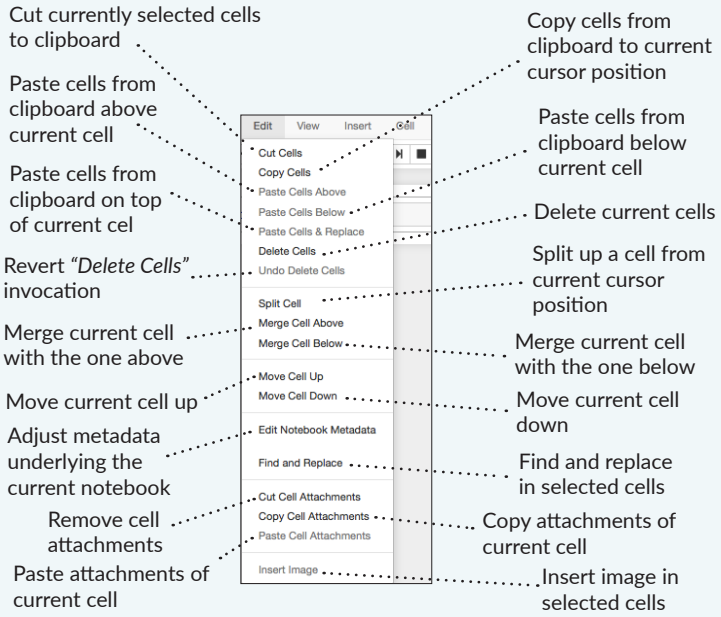
Saving/Loading Notebooks



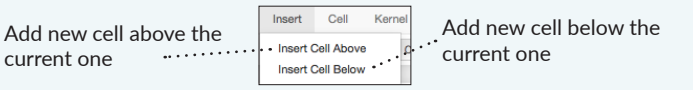
Writing Code And Text

Code and text are encapsulated by 3 basic cell types: markdown cells, code cells, and raw NBConvert cells.

Edit Cells



Insert Cells

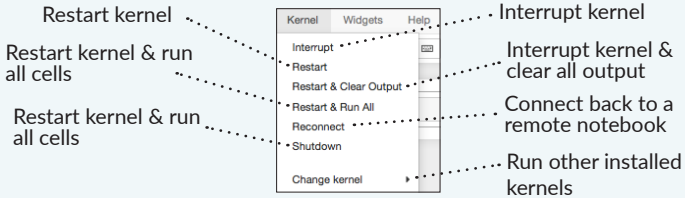


Working with Different Programming Languages

Kernels provide computation and communication with front-end interfaces like the notebooks. There are three main kernels:

IPython IRkernel IJulia

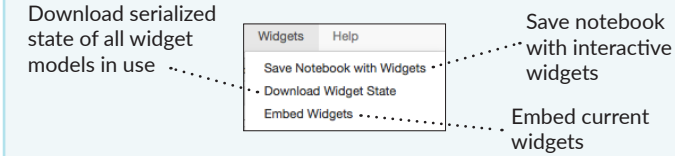
Installing Jupyter Notebook will automatically install the IPython kernel.



Widgets

Notebook widgets provide the ability to visualize and control changes in your data, often as a control like a slider, textbox, etc.

You can use them to build interactive GUIs for your notebooks or to synchronize stateful and stateless information between Python and JavaScript.

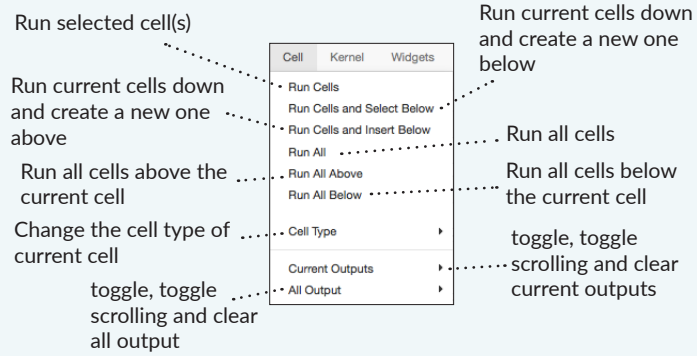


Command Mode:

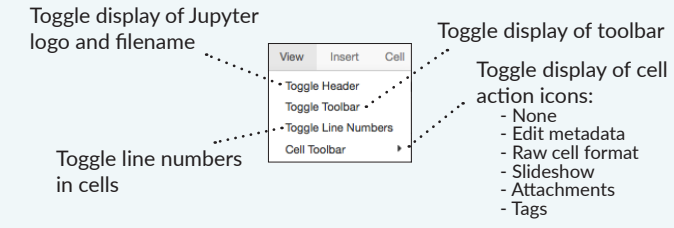
Edit Mode:

1. Save and checkpoint
2. Insert cell below
3. Cut cell
4. Copy cell(s)
5. Paste cell(s) below
6. Move cell up
7. Move cell down
8. Run current cell
9. Interrupt kernel
10. Restart kernel
11. Display characteristics
12. Open command palette
13. Current kernel
14. Kernel status
15. Log out from notebook server

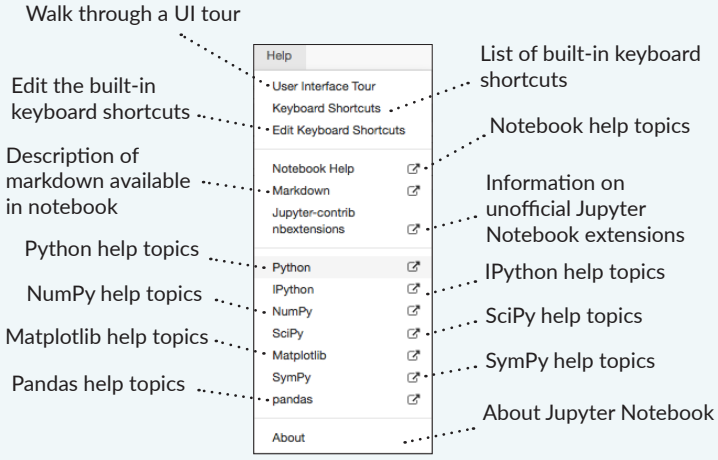
Executing Cells



View Cells



Asking For Help



Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

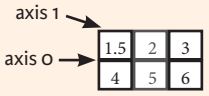
Use the following import convention:

```
>>> import numpy as np
```

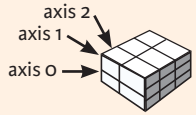
NumPy Arrays

1D array

2D array



3D array



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))                    Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16)    Create an array of ones
>>> d = np.arange(10,25,5)             Create an array of evenly
                                       spaced values (step value)
>>> np.linspace(0,2,9)                 Create an array of evenly
                                       spaced values (number of samples)
>>> e = np.full((2,2),7)               Create a constant array
>>> f = np.eye(2)                      Create a 2X2 identity matrix
>>> np.random.random((2,2))            Create an array with random values
>>> np.empty((3,2))                    Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64       Signed 64-bit integer types
>>> np.float32     Standard double-precision floating point
>>> np.complex     Complex numbers represented by 128 floats
>>> np.bool        Boolean type storing TRUE and FALSE values
>>> np.object      Python object type
>>> np.string_     Fixed-length string type
>>> np.unicode_    Fixed-length unicode type
```

Inspecting Your Array

```
>>> a.shape             Array dimensions
>>> len(a)              Length of array
>>> b.ndim              Number of array dimensions
>>> e.size              Number of array elements
>>> b.dtype             Data type of array elements
>>> b.dtype.name        Name of data type
>>> b.astype(int)       Convert an array to a different type
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b                          Subtraction
  array([[-0.5,  0. ,  0. ],
         [-3. , -3. , -3. ]])
>>> np.subtract(a,b)                   Subtraction
>>> b + a                              Addition
  array([[ 2.5,  4. ,  6. ],
         [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)                        Addition
>>> a / b                              Division
  array([[ 0.66666667,  1.        ,  1.        ],
         [ 0.25      ,  0.4       ,  0.5       ]])
>>> np.divide(a,b)                     Division
>>> a * b                              Multiplication
  array([[  1.5,   4. ,   9. ],
         [  4. ,  10. ,  18. ]])
>>> np.multiply(a,b)                   Multiplication
>>> np.exp(b)                          Exponentiation
>>> np.sqrt(b)                         Square root
>>> np.sin(a)                          Print sines of an array
>>> np.cos(b)                          Element-wise cosine
>>> np.log(a)                          Element-wise natural logarithm
>>> e.dot(f)                           Dot product
  array([[ 7.,  7.],
         [ 7.,  7.]])
```

Comparison

```
>>> a == b                             Element-wise comparison
  array([[False,  True,  True],
         [False, False, False]], dtype=bool)
>>> a < 2                              Element-wise comparison
  array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)               Array-wise comparison
```

Aggregate Functions

```
>>> a.sum()                            Array-wise sum
>>> a.min()                            Array-wise minimum value
>>> b.max(axis=0)                      Maximum value of an array row
>>> b.cumsum(axis=1)                   Cumulative sum of the elements
>>> a.mean()                           Mean
>>> b.median()                         Median
>>> a.corrcoef()                       Correlation coefficient
>>> np.std(b)                          Standard deviation
```

Copying Arrays

```
>>> h = a.view()          Create a view of the array with the same data
>>> np.copy(a)            Create a copy of the array
>>> h = a.copy()          Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort()              Sort an array
>>> c.sort(axis=0)        Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Also see Lists

Subsetting


```
>>> a[2]
  3
```

Select the element at the 2nd index


```
>>> b[1,2]
  6.0
```

Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing


```
>>> a[0:2]
  array([1, 2])
```

Select items at index 0 and 1


```
>>> b[0:2,1]
  array([ 2.,  5.])
```

Select items at rows 0 and 1 in column 1


```
>>> b[:1]
  array([[1.5, 2., 3.]])
```

Select all items at row 0 (equivalent to `b[0:1, :]`)

```
>>> c[1,...]
  array([[[ 3.,  2.,  1.],
          [ 4.,  5.,  6.]]])
```

Same as `[1,:,:]`

```
>>> a[ : :-1]
  array([3, 2, 1])
```

Reversed array `a`

Boolean Indexing


```
>>> a[a<2]
  array([1])
```

Select elements from `a` less than 2

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]
  array([ 4. , 2. , 6. , 1.5])
```

Select elements `(1,0)`, `(0,1)`, `(1,2)` and `(0,0)`

```
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]
  array([[ 4. , 5. , 6. , 4. ],
         [ 1.5, 2. , 3. , 1.5],
         [ 4. , 5. , 6. , 4. ],
         [ 1.5, 2. , 3. , 1.5]])
```

Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)        Permute array dimensions
>>> i.T                        Permute array dimensions
```

Changing Array Shape

```
>>> b.ravel()                  Flatten the array
>>> g.reshape(3,-2)            Reshape, but don't change data
```

Adding/Removing Elements

```
>>> h.resize((2,6))            Return a new array with shape (2,6)
>>> np.append(h,g)             Append items to an array
>>> np.insert(a, 1, 5)         Insert items in an array
>>> np.delete(a,[1])           Delete items from an array
```

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)        Concatenate arrays
  array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))                    Stack arrays vertically (row-wise)
  array([[ 1. ,  2. ,  3. ],
         [ 1.5,  2. ,  3. ],
         [ 4. ,  5. ,  6. ]])
>>> np.r_[e,f]                          Stack arrays vertically (row-wise)
>>> np.hstack((e,f))                    Stack arrays horizontally (column-wise)
  array([[ 7.,  7.,  1.,  0.],
         [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))              Create stacked column-wise arrays
  array([[ 1, 10],
         [ 2, 15],
         [ 3, 20]])
>>> np.c_[a,d]                          Create stacked column-wise arrays
```

Splitting Arrays

```
>>> np.hsplit(a,3)                      Split the array horizontally at the 3rd index
  [array([1]),array([2]),array([3])]
>>> np.vsplit(c,2)                      Split the array vertically at the 2nd index
[array([[[ 1.5,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]]),
 array([[[ 3.,  2.,  3.],
         [ 4.,  5.,  6.]]])]
```

Python For Data Science *Cheat Sheet*

SciPy - Linear Algebra

Learn More Python for Data Science Interactively at www.datacamp.com

SciPy

The **SciPy** library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.

Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j,2j,3j), (4j,5j,6j)])
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]])
```

Index Tricks

```
>>> np.mgrid[0:5,0:5]          Create a dense meshgrid
>>> np.ogrid[0:2,0:2]          Create an open meshgrid
>>> np.r_[[3,[0]*5,-1:1:10j]   Stack arrays vertically (row-wise)
>>> np.c_[b,c]                 Create stacked column-wise arrays
```

Shape Manipulation

```
>>> np.transpose(b)            Permute array dimensions
>>> b.flatten()                Flatten the array
>>> np.hstack((b,c))           Stack arrays horizontally (column-wise)
>>> np.vstack((a,b))           Stack arrays vertically (row-wise)
>>> np.hsplit(c,2)             Split the array horizontally at the 2nd index
>>> np.vpslit(d,2)             Split the array vertically at the 2nd index
```

Polynomials

```
>>> from numpy import poly1d
>>> p = poly1d([3,4,5])        Create a polynomial object
```

Vectorizing Functions

```
>>> def myfunc(a):
        if a < 0:
          return a*2
        else:
          return a/2
>>> np.vectorize(myfunc)       Vectorize functions
```

Type Handling

```
>>> np.real(c)                 Return the real part of the array elements
>>> np.imag(c)                 Return the imaginary part of the array elements
>>> np.real_if_close(c,tol=1000)  Return a real array if complex parts close to 0
>>> np.cast['f'](np.pi)        Cast object to a data type
```

Other Useful Functions

```
>>> np.angle(b,deg=True)       Return the angle of the complex argument
>>> g = np.linspace(0,np.pi,num=5)  Create an array of evenly spaced values
                                    (number of samples)
>>> g [3:] += np.pi
>>> np.unwrap(g)               Unwrap
>>> np.logspace(0,10,3)        Create an array of evenly spaced values (log scale)
>>> np.select([c<4],[c*2])     Return values from a list of arrays depending on
                               conditions
>>> misc.factorial(a)          Factorial
>>> misc.comb(10,3,exact=True)  Combine N things taken at k time
>>> misc.central_diff_weights(3)  Weights for Np-point central derivative
>>> misc.derivative(myfunc,1.0)   Find the n-th derivative of a function at a point
```

Linear Algebra

Also see NumPy

You'll use the `linalg` and `sparse` modules. Note that `scipy.linalg` contains and expands on `numpy.linalg`.

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Inverse

```
>>> A.I                        Inverse
>>> linalg.inv(A)              Inverse
>>> A.T                        Tranpose matrix
>>> A.H                        Conjugate transposition
>>> np.trace(A)                Trace
```

Norm

```
>>> linalg.norm(A)             Frobenius norm
>>> linalg.norm(A,1)           L1 norm (max column sum)
>>> linalg.norm(A,np.inf)      L inf norm (max row sum)
```

Rank

```
>>> np.linalg.matrix_rank(C)   Matrix rank
```

Determinant

```
>>> linalg.det(A)              Determinant
```

Solving linear problems

```
>>> linalg.solve(A,b)          Solver for dense matrices
>>> E = np.mat(a).T            Solver for dense matrices
>>> linalg.lstsq(D,E)          Least-squares solution to linear matrix
                               equation
```

Generalized inverse

```
>>> linalg.pinv(C)             Compute the pseudo-inverse of a matrix
                               (least-squares solver)
>>> linalg.pinv2(C)            Compute the pseudo-inverse of a matrix
                               (SVD)
```

Creating Sparse Matrices

```
>>> F = np.eye(3, k=1)         Create a 2X2 identity matrix
>>> G = np.mat(np.identity(2)) Create a 2x2 identity matrix
>>> C[C > 0.5] = 0
>>> H = sparse.csr_matrix(C)   Compressed Sparse Row matrix
>>> I = sparse.csc_matrix(D)   Compressed Sparse Column matrix
>>> J = sparse.dok_matrix(A)   Dictionary Of Keys matrix
>>> E.todense()                Sparse matrix to full matrix
>>> sparse.isspmatrix_csc(A)   Identify sparse matrix
```

Sparse Matrix Routines

Inverse

```
>>> sparse.linalg.inv(I)       Inverse
```

Norm

```
>>> sparse.linalg.norm(I)      Norm
```

Solving linear problems

```
>>> sparse.linalg.spsolve(H,I) Solver for sparse matrices
```

Sparse Matrix Functions

```
>>> sparse.linalg.expm(I)      Sparse matrix exponential
```

Matrix Functions

Addition

```
>>> np.add(A,D)                Addition
```

Subtraction

```
>>> np.subtract(A,D)           Subtraction
```

Division

```
>>> np.divide(A,D)             Division
```

Multiplication

```
>>> np.multiply(D,A)           Multiplication
>>> np.dot(A,D)                Dot product
>>> np.vdot(A,D)               Vector dot product
>>> np.inner(A,D)              Inner product
>>> np.outer(A,D)              Outer product
>>> np.tensordot(A,D)          Tensor dot product
>>> np.kron(A,D)               Kronecker product
```

Exponential Functions

```
>>> linalg.expm(A)             Matrix exponential
>>> linalg.expm2(A)            Matrix exponential (Taylor Series)
>>> linalg.expm3(D)            Matrix exponential (eigenvalue
                               decomposition)
```

Logarithm Function

```
>>> linalg.logm(A)             Matrix logarithm
```

Trigonometric Tunctions

```
>>> linalg.sinm(D)             Matrix sine
>>> linalg.cosm(D)             Matrix cosine
>>> linalg.tanm(A)             Matrix tangent
```

Hyperbolic Trigonometric Functions

```
>>> linalg.sinhm(D)            Hypberbolic matrix sine
>>> linalg.coshm(D)            Hyperbolic matrix cosine
>>> linalg.tanhm(A)            Hyperbolic matrix tangent
```

Matrix Sign Function

```
>>> np.sigm(A)                 Matrix sign function
```

Matrix Square Root

```
>>> linalg.sqrtm(A)            Matrix square root
```

Arbitrary Functions

```
>>> linalg.funm(A, lambda x: x*x)  Evaluate matrix function
```

Decompositions

Eigenvalues and Eigenvectors

```
>>> la, v = linalg.eig(A)      Solve ordinary or generalized
                               eigenvalue problem for square matrix
>>> l1, l2 = la                Unpack eigenvalues
>>> v[:,0]                     First eigenvector
>>> v[:,1]                     Second eigenvector
>>> linalg.eigvals(A)          Unpack eigenvalues
```

Singular Value Decomposition

```
>>> U,s,Vh = linalg.svd(B)     Singular Value Decomposition (SVD)
>>> M,N = B.shape
>>> Sig = linalg.diagsvd(s,M,N)  Construct sigma matrix in SVD
```

LU Decomposition

```
>>> P,L,U = linalg.lu(C)       LU Decomposition
```

Sparse Matrix Decompositions

```
>>> la, v = sparse.linalg.eigs(F,1)  Eigenvalues and eigenvectors
>>> sparse.linalg.svds(H, 2)         SVD
```

Asking For Help

```
>>> help(scipy.linalg.diagsvd)
>>> np.info(np.matrix)
```

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science Interactively at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

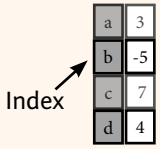
Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

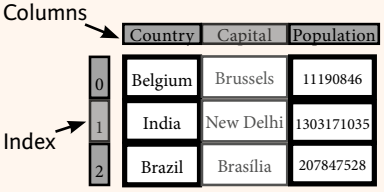
Series

A **one-dimensional** labeled array capable of holding any data type




```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
 -5

>>> df[1:]
   Country     Capital   Population
 1   India   New Delhi   1303171035
 2  Brazil    Brasília    207847528
```

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]]
 'Belgium'
>>> df.iat[[0],[0]]
 'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc[[0], ['Country']]
 'Belgium'
>>> df.at[[0], ['Country']]
 'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
 Country      Brazil
 Capital    Brasília
 Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:,'Capital']
 0     Brussels
 1    New Delhi
 2     Brasília
```

Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
 'New Delhi'
```

Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population']>1200000000]
```

Series `s` where value is not >1
`s` where value is <-1 or >2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```

Set index `a` of Series `s` to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis
Sort by the values along an axis
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idxmin()/df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
  a     10.0
  b      NaN
  c      5.0
  d      7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
  a    10.0
  b    -5.0
  c    5.0
  d    7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> pd.to_sql('myDf', engine)
```

Python For Data Science *Cheat Sheet*

Scikit-Learn

Learn Python for data science Interactively at www.DataCamp.com

Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.

A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test= train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M','M','F','F','M','F','M','M','F','F','F'])
>>> X[X < 0.7] = 0
```

Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
                                                        y,
                                                        random_state=0)
```

Create Your Model

Supervised Learning Estimators

Linear Regression

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

Support Vector Machines (SVM)

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

Naive Bayes

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

KNN

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

Unsupervised Learning Estimators

Principal Component Analysis (PCA)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

K Means

```
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

Model Fitting

Supervised learning	
`>>> lr.fit(X, y)`	Fit the model to the data
`>>> knn.fit(X_train, y_train)`	
`>>> svc.fit(X_train, y_train)`	
Unsupervised Learning	
`>>> k_means.fit(X_train)`	Fit the model to the data
`>>> pca_model = pca.fit_transform(X_train)`	Fit to data, then transform it

Prediction

Supervised Estimators	
`>>> y_pred = svc.predict(np.random.random((2,5)))`	Predict labels
`>>> y_pred = lr.predict(X_test)`	Predict labels
`>>> y_pred = knn.predict_proba(X_test)`	Estimate probability of a label
Unsupervised Estimators	
`>>> y_pred = k_means.predict(X_test)`	Predict labels in clustering algos

Preprocessing The Data

Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

Encoding Categorical Features

```
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> y = enc.fit_transform(y)
```

Imputing Missing Values

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit_transform(X_train)
```

Generating Polynomial Features

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(5)
>>> poly.fit_transform(X)
```

Evaluate Your Model's Performance

Classification Metrics

Accuracy Score

```
>>> knn.score(X_test, y_test)
```
Estimator score method

```
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
```
Metric scoring functions

Classification Report

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, y_pred))
```
Precision, recall, f1-score and support

Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

Regression Metrics

Mean Absolute Error

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

Mean Squared Error

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

R^2 Score

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

Clustering Metrics

Adjusted Rand Index

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

Homogeneity

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

V-measure

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

Tune Your Model

Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
              "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
                        param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5),
              "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn,
                                 param_distributions=params,
                                 cv=4,
                                 n_iter=8,
                                 random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```

Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python Interactively at www.DataCamp.com

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

matplotlib

1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

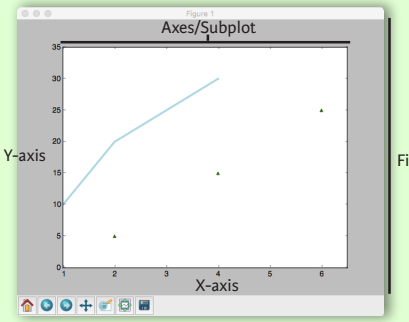
Axes

All plotting is done with respect to an `Axes`. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data 2 Create plot 3 Plot 4 Customize plot 5 Save plot 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]                          Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure()                     Step 2
>>> ax = fig.add_subplot(111)              Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3)   Step 3, 4
>>> ax.scatter([2,4,6],
               [5,15,25],
               color='darkgreen',
               marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()                             Step 6
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                   cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(8, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)                  Add padding to a plot
>>> ax.axis('equal')                         Set the aspect ratio of the plot to 1
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])    Set limits for x-and y-axis
>>> ax.set_xlim(0,10.5)                      Set limits for x-axis
```

Legends

```
>>> ax.set(title='An Example Axes',          Set a title and x-and y-axis labels
           ylabel='Y-Axis',
           xlabel='X-Axis')
>>> ax.legend(loc='best')                    No overlapping plot elements
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),           Manually set x-ticks
                 ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',                 Make y-ticks longer and go in and out
                   direction='inout',
                   length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,         Adjust the spacing between subplots
                         hspace=0.3,
                         left=0.125,
                         right=0.9,
                         top=0.9,
                         bottom=0.1)
>>> fig.tight_layout()                       Fit subplot(s) in to the figure area
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)     Make the top axis line for a plot invisible
>>> ax1.spines['bottom'].set_position(('outward',10))   Move the bottom axis line outward
```

3 Plotting Routines

1D Data

```
>>> fig, ax = plt.subplots()
>>> lines = ax.plot(x,y)                     Draw points with lines or markers connecting them
>>> ax.scatter(x,y)                          Draw unconnected points, scaled or colored
>>> axes[0,0].bar([1,2,3],[3,4,5])           Plot vertical rectangles (constant width)
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])      Plot horiontal rectangles (constant height)
>>> axes[1,1].axhline(0.45)                  Draw a horizontal line across axes
>>> axes[0,1].axvline(0.65)                  Draw a vertical line across axes
>>> ax.fill(x,y,color='blue')                Draw filled polygons
>>> ax.fill_between(x,y,color='yellow')      Fill between y-values and o
```

Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)             Add an arrow to the axes
>>> axes[1,1].quiver(y,z)                    Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V)            Plot a 2D field of arrows
```

Data Distributions

```
>>> ax1.hist(y)                              Plot a histogram
>>> ax3.boxplot(y)                           Make a box and whisker plot
>>> ax3.violinplot(z)                        Make a violin plot
```

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,                      Colormapped or RGB arrays
                   cmap='gist_earth',
                   interpolation='nearest',
                   vmin=-2,
                   vmax=2)
```

```
>>> axes2[0].pcolor(data2)                   Pseudocolor plot of 2D array
>>> axes2[0].pcolormesh(data)                Pseudocolor plot of 2D array
>>> CS = plt.contour(Y,X,U)                  Plot contours
>>> axes2[2].contourf(data1)                 Plot filled contours
>>> axes2[2]= ax.clabel(CS)                  Label a contour plot
```

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

```
>>> plt.cla()                                Clear an axis
>>> plt.clf()                                Clear the entire figure
>>> plt.close()                              Close a window
```

Python For Data Science *Cheat Sheet*

Seaborn

Learn Data Science Interactively at www.DataCamp.com

Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on matplotlib and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

1. Prepare some data
2. Control figure aesthetics
3. Plot with Seaborn
4. Further customize your plot

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")                    Step 1
>>> sns.set_style("whitegrid")                         Step 2
>>> g = sns.lmplot(x="tip",                            Step 3
                   y="total_bill",
                   data=tips,
                   aspect=2)
>>> g = (g.set_axis_labels("Tip","Total bill(USD)").
set(xlim=(0,10),ylim=(0,100)))                         Step 4
>>> plt.title("title")
>>> plt.show(g)                                        Step 5
```

1 Data

Also see Lists, NumPy & Pandas

```
>>> import pandas as pd
>>> import numpy as np
>>> uniform_data = np.random.rand(10, 12)
>>> data = pd.DataFrame({'x':np.arange(1,101),
                         'y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")
>>> iris = sns.load_dataset("iris")
```

2 Figure Aesthetics

Also see Matplotlib

```
>>> f, ax = plt.subplots(figsize=(5,6))
```
Create a figure and one subplot

Seaborn styles

```
>>> sns.set()
>>> sns.set_style("whitegrid")
>>> sns.set_style("ticks",
                  {"xtick.major.size":8,
                   "ytick.major.size":8})
>>> sns.axes_style("whitegrid")
```
(Re)set the seaborn default
Set the matplotlib parameters
Set the matplotlib parameters

Return a dict of params or use with `with` to temporarily set the style

Context Functions

```
>>> sns.set_context("talk")
>>> sns.set_context("notebook",
                    font_scale=1.5,
                    rc={"lines.linewidth":2.5})
```
Set context to `"talk"`
Set context to `"notebook"`, scale font elements and override param mapping

Color Palette

```
>>> sns.set_palette("husl",3)
>>> sns.color_palette("husl")
>>> flatui = ["#9b59b6","#3498db","#95a5a6","#e74c3c","#34495e","#2ecc71"]
>>> sns.set_palette(flatui)
```
Define the color palette
Use with `with` to temporarily set palette
Set your own color palette

3 Plotting With Seaborn

Axis Grids

```
>>> g = sns.FacetGrid(titanic,
                      col="survived",
                      row="sex")
>>> g = g.map(plt.hist,"age")
```
Subplot grid for plotting conditional relationships

```
>>> sns.factorplot(x="pclass",
                   y="survived",
                   hue="sex",
                   data=titanic)
```
Draw a categorical plot onto a Facetgrid

```
>>> sns.lmplot(x="sepal_width",
               y="sepal_length",
               hue="species",
               data=iris)
```
Plot data and regression model fits across a FacetGrid

```
>>> h = sns.PairGrid(iris)
>>> h = h.map(plt.scatter)
>>> sns.pairplot(iris)
```
Subplot grid for plotting pairwise relationships
Plot pairwise bivariate distributions

```
>>> i = sns.JointGrid(x="x",
                      y="y",
                      data=data)
>>> i = i.plot(sns.regplot,
               sns.distplot)
```
Grid for bivariate plot with marginal univariate plots

```
>>> sns.jointplot("sepal_length",
                  "sepal_width",
                  data=iris,
                  kind='kde')
```
Plot bivariate distribution

Categorical Plots

Scatterplot

```
>>> sns.stripplot(x="species",
                  y="petal_length",
                  data=iris)
```
Scatterplot with one categorical variable

```
>>> sns.swarmplot(x="species",
                  y="petal_length",
                  data=iris)
```
Categorical scatterplot with non-overlapping points

Bar Chart

```
>>> sns.barplot(x="sex",
                y="survived",
                hue="class",
                data=titanic)
```
Show point estimates and confidence intervals with scatterplot glyphs

Count Plot

```
>>> sns.countplot(x="deck",
                  data=titanic,
                  palette="Greens_d")
```
Show count of observations

Point Plot

```
>>> sns.pointplot(x="class",
                  y="survived",
                  hue="sex",
                  data=titanic,
                  palette={"male":"g",
                           "female":"m"},
                  markers=["^","o"],
                  linestyles=["-","--"])
```
Show point estimates and confidence intervals as rectangular bars

Boxplot

```
>>> sns.boxplot(x="alive",
                y="age",
                hue="adult_male",
                data=titanic)
```
Boxplot

```
>>> sns.boxplot(data=iris,orient="h")
```
Boxplot with wide-form data

Violinplot

```
>>> sns.violinplot(x="age",
                   y="sex",
                   hue="survived",
                   data=titanic)
```
Violin plot

Regression Plots

```
>>> sns.regplot(x="sepal_width",
                y="sepal_length",
                data=iris,
                ax=ax)
```
Plot data and a linear regression model fit

Distribution Plots

```
>>> plot = sns.distplot(data.y,
                        kde=False,
                        color="b")
```
Plot univariate distribution

Matrix Plots

```
>>> sns.heatmap(uniform_data,vmin=0,vmax=1)
```
Heatmap

4 Further Customizations

Also see Matplotlib

Axisgrid Objects

```
>>> g.despine(left=True)
>>> g.set_ylabels("Survived")
>>> g.set_xticklabels(rotation=45)
>>> g.set_axis_labels("Survived",
                      "Sex")
```
Remove left spine
Set the labels of the y-axis
Set the tick labels for x
Set the axis labels

```
>>> h.set(xlim=(0,5),
          ylim=(0,5),
          xticks=[0,2.5,5],
          yticks=[0,2.5,5])
```
Set the limit and ticks of the x-and y-axis

Plot

```
>>> plt.title("A Title")
>>> plt.ylabel("Survived")
>>> plt.xlabel("Sex")
>>> plt.ylim(0,100)
>>> plt.xlim(0,10)
>>> plt.setp(ax,yticks=[0,5])
>>> plt.tight_layout()
```
Add plot title
Adjust the label of the y-axis
Adjust the label of the x-axis
Adjust the limits of the y-axis
Adjust the limits of the x-axis
Adjust a plot property
Adjust subplot params

5 Show or Save Plot

Also see Matplotlib

```
>>> plt.show()
>>> plt.savefig("foo.png")
>>> plt.savefig("foo.png",
                transparent=True)
```
Show the plot
Save the plot as a figure
Save transparent figure

Close & Clear

Also see Matplotlib

```
>>> plt.cla()
>>> plt.clf()
>>> plt.close()
```
Clear an axis
Clear an entire figure
Close a window

Python For Data Science *Cheat Sheet*

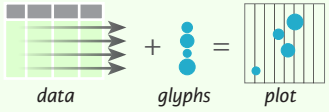
Bokeh

Learn Bokeh Interactively at www.DataCamp.com, taught by Bryan Van de Ven, core contributor

Plotting With Bokeh

The Python interactive visualization library **Bokeh** enables high-performance visual presentation of large datasets in modern web browsers.

Bokeh's mid-level general purpose `bokeh.plotting` interface is centered around two main components: data and glyphs.



The basic steps to creating plots with the `bokeh.plotting` interface are:

1. Prepare some data:

Python lists, NumPy arrays, Pandas DataFrames and other sequences of values

2. Create a new plot
3. Add renderers for your data, with visual customizations
4. Specify where to generate the output
5. Show or save the results

```
>>> from bokeh.plotting import figure
>>> from bokeh.io import output_file, show
>>> x = [1, 2, 3, 4, 5]                                  Step 1
>>> y = [6, 7, 2, 4, 5]
>>> p = figure(title="simple line example",               Step 2
              x_axis_label='x',
              y_axis_label='y')
>>> p.line(x, y, legend="Temp.", line_width=2)           Step 3
>>> output_file("lines.html")                            Step 4
>>> show(p)                                              Step 5
```

1 Data

Also see Lists, NumPy & Pandas

Under the hood, your data is converted to Column Data Sources. You can also do this manually:

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(np.array([[33.9,4,65, 'US'],
                                [32.4,4,66, 'Asia'],
                                [21.4,4,109, 'Europe']]),
                      columns=['mpg','cyl', 'hp', 'origin'],
                      index=['Toyota', 'Fiat', 'Volvo'])
```

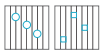
```
>>> from bokeh.models import ColumnDataSource
>>> cds_df = ColumnDataSource(df)
```

2 Plotting

```
>>> from bokeh.plotting import figure
>>> p1 = figure(plot_width=300, tools='pan,box_zoom')
>>> p2 = figure(plot_width=300, plot_height=300,
                x_range=(0, 8), y_range=(0, 8))
>>> p3 = figure()
```

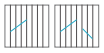
3 Renderers & Visual Customizations

Glyphs



Scatter Markers

```
>>> p1.circle(np.array([1,2,3]), np.array([3,2,1]),
              fill_color='white')
>>> p2.square(np.array([1.5,3.5,5.5]), [1,4,3],
              color='blue', size=1)
```



Line Glyphs

```
>>> p1.line([1,2,3,4], [3,4,5,6], line_width=2)
>>> p2.multi_line(pd.DataFrame([[1,2,3],[5,6,7]]),
                  pd.DataFrame([[3,4,5],[3,2,1]]),
                  color="blue")
```

Customized Glyphs

Also see Data

Selection and Non-Selection Glyphs

```
>>> p = figure(tools='box_select')
>>> p.circle('mpg', 'cyl', source=cds_df,
             selection_color='red',
             nonselection_alpha=0.1)
```

Hover Glyphs

```
>>> from bokeh.models import HoverTool
>>> hover = HoverTool(tooltips=None, mode='vline')
>>> p3.add_tools(hover)
```

Colormapping

```
>>> from bokeh.models import CategoricalColorMapper
>>> color_mapper = CategoricalColorMapper(
                   factors=['US', 'Asia', 'Europe'],
                   palette=['blue', 'red', 'green'])
>>> p3.circle('mpg', 'cyl', source=cds_df,
             color=dict(field='origin',
                        transform=color_mapper),
             legend='Origin')
```

Legend Location

Inside Plot Area

```
>>> p.legend.location = 'bottom_left'
```

Outside Plot Area

```
>>> from bokeh.models import Legend
>>> r1 = p2.asterisk(np.array([1,2,3]), np.array([3,2,1])
>>> r2 = p2.line([1,2,3,4], [3,4,5,6])
>>> legend = Legend(items=[("One" ,[p1, r1]),("Two",[r2])],
                    location=(0, -30))
>>> p.add_layout(legend, 'right')
```

Legend Orientation

```
>>> p.legend.orientation = "horizontal"
>>> p.legend.orientation = "vertical"
```

Legend Background & Border

```
>>> p.legend.border_line_color = "navy"
>>> p.legend.background_fill_color = "white"
```

Rows & Columns Layout

Rows

```
>>> from bokeh.layouts import row
>>> layout = row(p1,p2,p3)
```

Columns

```
>>> from bokeh.layouts import columns
>>> layout = column(p1,p2,p3)
```

Nesting Rows & Columns

```
>>>layout = row(column(p1,p2), p3)
```

Grid Layout

```
>>> from bokeh.layouts import gridplot
>>> row1 = [p1,p2]
>>> row2 = [p3]
>>> layout = gridplot([[p1,p2],[p3]])
```

Tabbed Layout

```
>>> from bokeh.models.widgets import Panel, Tabs
>>> tab1 = Panel(child=p1, title="tab1")
>>> tab2 = Panel(child=p2, title="tab2")
>>> layout = Tabs(tabs=[tab1, tab2])
```

Linked Plots

Linked Axes

```
>>> p2.x_range = p1.x_range
>>> p2.y_range = p1.y_range
```

Linked Brushing

```
>>> p4 = figure(plot_width = 100,
                tools='box_select,lasso_select')
>>> p4.circle('mpg', 'cyl', source=cds_df)
>>> p5 = figure(plot_width = 200,
                tools='box_select,lasso_select')
>>> p5.circle('mpg', 'hp', source=cds_df)
>>> layout = row(p4,p5)
```

4 Output & Export

Notebook

```
>>> from bokeh.io import output_notebook, show
>>> output_notebook()
```

HTML

Standalone HTML

```
>>> from bokeh.embed import file_html
>>> from bokeh.resources import CDN
>>> html = file_html(p, CDN, "my_plot")
```

```
>>> from bokeh.io import output_file, show
>>> output_file('my_bar_chart.html', mode='cdn')
```

Components

```
>>> from bokeh.embed import components
>>> script, div = components(p)
```

PNG

```
>>> from bokeh.io import export_png
>>> export_png(p, filename="plot.png")
```

SVG

```
>>> from bokeh.io import export_svgs
>>> p.output_backend = "svg"
>>> export_svgs(p, filename="plot.svg")
```

5 Show or Save Your Plots

```
>>> show(p1)
>>> save(p1)
```

```
>>> show(layout)
>>> save(layout)
```

Cheat Sheet: The pandas DataFrame Object

Preliminaries

Always start by importing these Python modules

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas import DataFrame, Series
```

Note: these are the recommended import aliases
Note: you can put these into a PYTHONSTARTUP file

Cheat sheet conventions

Code examples

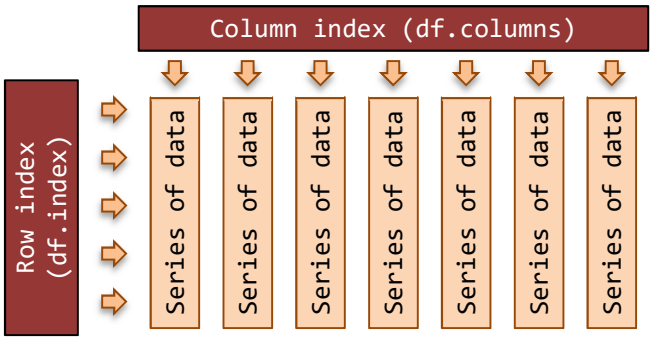
```
# Code examples are found in yellow boxes
```

In the code examples, typically I use:

- s to represent a pandas Series object;
- df to represent a pandas DataFrame object;
- idx to represent a pandas Index object.
- Also: t – tuple, l – list, b – Boolean, i – integer, a – numpy array, st – string, d – dictionary, etc.

The conceptual model

DataFrame object: is a two-dimensional table of data with column and row indexes (something like a spread sheet). The columns are made up of Series objects.



A DataFrame has two Indexes:

- Typically, the column index (df.columns) is a list of strings (variable names) or (less commonly) integers
- Typically, the row index (df.index) might be:
  - Integers - for case or row numbers;
  - Strings – for case names; or
  - DatetimeIndex or PeriodIndex – for time series

Series object: an ordered, one-dimensional array of data with an index. All the data in a Series is of the same data type. Series arithmetic is vectorised after first aligning the Series index for each of the operands.

```
s1 = Series(range(0,4))  # -> 0, 1, 2, 3
s2 = Series(range(1,5))  # -> 1, 2, 3, 4
s3 = s1 + s2             # -> 1, 3, 5, 7
```

Get your data into a DataFrame

Instantiate an empty DataFrame

```
df = DataFrame()
```

Load a DataFrame from a CSV file

```
df = pd.read_csv('file.csv') # often works
df = pd.read_csv('file.csv', header=0,
    index_col=0, quotechar='"', sep=':',
    na_values = ['na', '-', '.', ''])
```

Note: refer to pandas docs for all arguments

Get data from inline CSV text to a DataFrame

```
from io import StringIO
data = """, Animal, Cuteness, Desirable
row-1,      dog,    8.7,   True
row-2,      cat,    9.5,   True
row-3,      bat,    2.6,   False"""
df = pd.read_csv(StringIO(data), header=0,
    index_col=0, skipinitialspace=True)
```

Note: *skipinitialspace=True* allows for a pretty layout

Load DataFrames from a Microsoft Excel file

```
# Each Excel sheet in a Python dictionary
workbook = pd.ExcelFile('file.xlsx')
d = {} # start with an empty dictionary
for sheet_name in workbook.sheet_names:
    df = workbook.parse(sheet_name)
    d[sheet_name] = df
```

Note: the parse() method takes many arguments like read_csv() above. Refer to the pandas documentation.

Load a DataFrame from a MySQL database

```
import pymysql
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://'
    +'USER:PASSWORD@HOST/DATABASE')
df = pd.read_sql_table('table', engine)
```

Data in Series then combine into a DataFrame

```
# Example 1 ...
s1 = Series(range(6))
s2 = s1 * s1
s2.index = s2.index + 2 # misalign indexes
df = pd.concat([s1, s2], axis=1)

# Example 2 ...
s3 = Series({'Tom':1, 'Dick':4, 'Har':9})
s4 = Series({'Tom':3, 'Dick':2, 'Mar':5})
df = pd.concat({'A':s3, 'B':s4 }, axis=1)
```

Note: 1st method has in integer column labels
Note: 2nd method does not guarantee col order

Get a DataFrame from a Python dictionary

```
# default --- assume data is in columns
df = DataFrame({
        'col0' : [1.0, 2.0, 3.0, 4.0],
        'col1' : [100, 200, 300, 400]
    })
```

Get a DataFrame from data in a Python dictionary

```
# --- use helper method for data in rows
df = DataFrame.from_dict({ # data by row
        # rows as python dictionaries
        'row0' : {'col0':0, 'col1':'A'},
        'row1' : {'col0':1, 'col1':'B'}
    }, orient='index')

df = DataFrame.from_dict({ # data by row
        # rows as python lists
        'row0' : [1, 1+1j, 'A'],
        'row1' : [2, 2+2j, 'B']
    }, orient='index')
```

Create play/fake data (useful for testing)

```
# --- simple - default integer indexes
df = DataFrame(np.random.rand(50,5))

# --- with a time-stamp row index:
df = DataFrame(np.random.rand(500,5))
df.index = pd.date_range('1/1/2005',
       periods=len(df), freq='M')

# --- with alphabetic row and col indexes
#          and a "groupable" variable
import string
import random
rows = 52
cols = 5
assert(1 <= rows <= 52) # min/max row count
df = DataFrame(np.random.randn(rows, cols),
   columns=['c'+str(i) for i in range(cols)],
   index=list((string.ascii_uppercase +
         string.ascii_lowercase)[0:rows]))
df['groupable'] = [random.choice('abcde')
    for _ in range(rows)]
```

Saving a DataFrame

Saving a DataFrame to a CSV file

```
df.to_csv('name.csv', encoding='utf-8')
```

Saving DataFrames to an Excel Workbook

```
from pandas import ExcelWriter
writer = ExcelWriter('filename.xlsx')
df1.to_excel(writer,'Sheet1')
df2.to_excel(writer,'Sheet2')
writer.save()
```

Saving a DataFrame to MySQL

```
import pymysql
from sqlalchemy import create_engine
e = create_engine('mysql+pymysql://' +
    'USER:PASSWORD@HOST/DATABASE')
df.to_sql('TABLE',e, if_exists='replace')
```

Note: if_exists → 'fail', 'replace', 'append'

Saving to Python objects

```
d = df.to_dict()        # to dictionary
str = df.to_string()    # to string
m = df.as_matrix()      # to numpy matrix
```

Working with the whole DataFrame

Peek at the DataFrame contents/structure

```
df.info()           # index & data types
dfh = df.head(i)    # get first i rows
dft = df.tail(i)    # get last i rows
dfs = df.describe() # summary stats cols
top_left_corner_df = df.iloc[:4, :4]
```

DataFrame non-indexing attributes

```
df = df.T       # transpose rows and cols
l = df.axes     # list row and col indexes
(r_idx, c_idx) = df.axes      # from above
s = df.dtypes   # Series column data types
b = df.empty    # True for empty DataFrame
i = df.ndim     # number of axes (it is 2)
t = df.shape    # (row-count, column-count)
i = df.size     # row-count * column-count
a = df.values   # get a numpy array for df
```

DataFrame utility methods

```
df = df.copy()  # copy a DataFrame
df = df.rank()  # rank each col (default)
df = df.sort_values(by=col)
df = df.sort_values(by=[col1, col2])
df = df.sort_index()
df = df.astype(dtype) # type conversion
```

DataFrame iteration methods

```
df.iteritems()  # (col-index, Series) pairs
df.iterrows()   # (row-index, Series) pairs
# example ... iterating over columns ...
for (name, series) in df.iteritems():
    print('\nCol name: ' + str(name))
    print('1st value: ' + str(series.iat[0]))
```

Maths on the whole DataFrame (not a complete list)

```
df = df.abs()     # absolute values
df = df.add(o)    # add df, Series or value
s =  df.count()   # non NA/null values
df = df.cummax()  # (cols default axis)
df = df.cummin()  # (cols default axis)
df = df.cumsum()  # (cols default axis)
df = df.diff()    # 1st diff (col def axis)
df = df.div(o)    # div by df, Series, value
df = df.dot(o)    # matrix dot product
s =  df.max()     # max of axis (col def)
s =  df.mean()    # mean (col default axis)
s =  df.median()  # median (col default)
s =  df.min()     # min of axis (col def)
df = df.mul(o)    # mul by df Series val
s =  df.sum()     # sum axis (cols default)
df = df.where(df > 0.5, other=np.nan)
```

Note: methods returning a series default to work on cols

Select/filter rows/cols based on index label values

```
df = df.filter(items=['a', 'b'])  # by col
df = df.filter(items=[5], axis=0) # by row
df = df.filter(like='x') # keep x in col
df = df.filter(regex='x') # regex in col
df = df.select(lambda x: not x%5) # 5th rows
```

Note: select takes a Boolean function, for cols: axis=1
Note: filter defaults to cols; select defaults to rows

Working with Columns

Get column index and labels

```
idx = df.columns        # get col index
label = df.columns[0]   # first col label
l = df.columns.tolist() # list of col labels
a = df.columns.values   # array of col labels
```

Change column labels

```
df = df.rename(columns={'old':'new','a':'1'})
df.columns = ['new1', 'new2', 'new3'] # etc.
```

Selecting columns

```
s =  df['colName']     # select col to Series
df = df[['colName']]   # select col to df
df = df[['a','b']]     # select 2-plus cols
df = df[['c','a','b']] # change col order
s =  df[df.columns[0]] # select by number
df = df[df.columns[[0, 3, 4]]] # by numbers
df = [df.columns[:-1]] # all but last col
s =  df.pop('c')       # get & drop from df
```

Selecting columns with Python attributes

```
s = df.a # same as s = df['a']
# cannot create new columns by attribute
df.existing_column = df.a / df.b
df['new_column'] = df.a / df.b
```

Trap: column names must be valid identifiers.

Adding new columns to a DataFrame

```
df['new_col'] = range(len(df))
df['new_col'] = np.repeat(np.nan,len(df))
df['random'] = np.random.rand(len(df))
df['index_as_col'] = df.index
df1[['b','c']] = df2[['e','f']]
df3 = df1.append(other=df2)
```

Trap: When adding a new column, only items from the new series that have a corresponding index in the DataFrame will be added. The receiving DataFrame is **not** extended to accommodate the new series.
Trap: when adding a python list or numpy array, the column will be added by integer position.

Swap column contents

```
df[['B', 'A']] = df[['A', 'B']]
```

Dropping (deleting) columns (mostly by label)

```
df = df.drop('col1', axis=1)
df.drop('col1', axis=1, inplace=True)
df = df.drop(['col1','col2'], axis=1)
s = df.pop('col') # drops from frame
del df['col'] # even classic python works
df = df.drop(df.columns[0], axis=1)#first
df = df.drop(df.columns[-1:],axis=1)#last
```

Vectorised arithmetic on columns

```
df['proportion']=df['count']/df['total']
df['percent'] = df['proportion'] * 100.0
```

Apply numpy mathematical functions to columns

```
df['log_data'] = np.log(df['col1'])
```

Note: many many more numpy math functions
Hint: Prefer pandas math over numpy where you can.

Set column values set based on criteria

```
df['b'] = df['a'].where(df['a']>0, other=0)
df['d'] = df['a'].where(df.b!=0, other=df.c)
```

Note: where other can be a Series or a scalar

Data type conversions

```
st =  df['col'].astype(str)# Series dtype
a = df['col'].values       # numpy array
l = df['col'].tolist()     # python list
```

Note: useful dtypes for Series conversion: int, float, str
Trap: index lost in conversion from Series to array or list

Common column-wide methods/attributes

```
value = df['col'].dtype    # type of data
value = df['col'].size     # col dimensions
value = df['col'].count()  # non-NA count
value = df['col'].sum()
value = df['col'].prod()
value = df['col'].min()
value = df['col'].max()
value = df['col'].mean()   # also median()
value = df['col'].cov(df['col2'])
s =     df['col'].describe()
s =     df['col'].value_counts()
```

Find index label for min/max values in column

```
label =  df['col1'].idxmin()
label =  df['col1'].idxmax()
```

Common column element-wise methods

```
s = df['col'].isnull()
s = df['col'].notnull()    # not isnull()
s = df['col'].astype(float)
s = df['col'].abs()
s = df['col'].round(decimals=0)
s = df['col'].diff(periods=1)
s = df['col'].shift(periods=1)
s = df['col'].to_datetime()
s = df['col'].fillna(0)    # replace NaN w 0
s = df['col'].cumsum()
s = df['col'].cumprod()
s = df['col'].pct_change(periods=4)
s = df['col'].rolling(window=4,
    min_periods=4, center=False).sum()
```

Append a column of row sums to a DataFrame

```
df['Total'] = df.sum(axis=1)
```

Note: also means, mins, maxs, etc.

Multiply every column in DataFrame by Series

```
df = df.mul(s, axis=0) # on matched rows
```

Note: also add, sub, div, etc.

Selecting columns with .loc, .iloc and .ix

```
df = df.loc[:, 'col1':'col2'] # inclusive
df = df.iloc[:, 0:2]          # exclusive
```

Get the integer position of a column index label

```
i = df.columns.get_loc('col_name')
```

Test if column index values are unique/monotonic

```
if df.columns.is_unique: pass # ...
b = df.columns.is_monotonic_increasing
b = df.columns.is_monotonic_decreasing
```

Working with rows

Get the row index and labels

```
idx = df.index          # get row index
label = df.index[0]     # first row label
label = df.index[-1]    # last row label
l = df.index.tolist()   # get as a list
a = df.index.values     # get as an array
```

Change the (row) index

```
df.index = idx            # new ad hoc index
df = df.set_index('A')    # col A new index
df = df.set_index(['A', 'B']) # MultiIndex
df = df.reset_index()     # replace old w new
# note: old index stored as a col in df
df.index = range(len(df)) # set with list
df = df.reindex(index=range(len(df)))
df = df.set_index(keys=['r1','r2','etc'])
df.rename(index={'old':'new'}, inplace=True)
```

Adding rows

```
df = original_df.append(more_rows_in_df)
```

Hint: convert row to a DataFrame and then append.
Both DataFrames should have same column labels.

Dropping rows (by name)

```
df = df.drop('row_label')
df = df.drop(['row1','row2']) # multi-row
```

Boolean row selection by values in a column

```
df = df[df['col2'] >= 0.0]
df = df[(df['col3']>=1.0) | (df['col1']<0.0)]
df = df[df['col'].isin([1,2,5,7,11])]
df = df[~df['col'].isin([1,2,5,7,11])]
df = df[df['col'].str.contains('hello')]
```

Trap: bitwise "or", "and" “not; (ie. | & ~) co-opted to be Boolean operators on a Series of Boolean
Trap: need parentheses around comparisons.

Selecting rows using isin over multiple columns

```
# fake up some data
data = {1:[1,2,3], 2:[1,4,9], 3:[1,8,27]}
df = DataFrame(data)

# multi-column isin
lf = {1:[1, 3], 3:[8, 27]} # look for
f = df[df[list(lf)].isin(lf).all(axis=1)]
```

Selecting rows using an index

```
idx = df[df['col'] >= 2].index
print(df.ix[idx])
```

Select a slice of rows by integer position
[inclusive-from : exclusive-to [: step]]
start is 0; end is len(df)

```
df = df[:]       # copy entire DataFrame
df = df[0:2]     # rows 0 and 1
df = df[2:3]     # row 2 (the third row)
df = df[-1:]     # the last row
df = df[:-1]     # all but the last row
df = df[::2]     # every 2nd row (0 2 ..)
```

Trap: a single integer without a colon is a column label for integer numbered columns.

Select a slice of rows by label/index
[inclusive-from : inclusive–to [: step]]

```
df = df['a':'c'] # rows 'a' through 'c'
```

Trap: cannot work for integer labelled rows – see previous code snippet on integer position slicing.

Append a row of column totals to a DataFrame

```
# Option 1: use dictionary comprehension
sums = {col: df[col].sum() for col in df}
sums_df = DataFrame(sums,index=['Total'])
df = df.append(sums_df)

# Option 2: All done with pandas
df = df.append(DataFrame(df.sum(),
                  columns=['Total']).T)
```

Iterating over DataFrame rows

```
for (index, row) in df.iterrows(): # pass
```

Trap: row data type may be coerced.

Sorting DataFrame rows values

```
df = df.sort(df.columns[0],
                         ascending=False)
df.sort(['col1', 'col2'], inplace=True)
```

Sort DataFrame by its row index

```
df.sort_index(inplace=True) # sort by row
df = df.sort_index(ascending=False)
```

Random selection of rows

```
import random as r
k = 20 # pick a number
selection = r.sample(range(len(df)), k)
df_sample = df.iloc[selection, :] # get copy
```

Note: this randomly selected sample is not sorted

Drop duplicates in the row index

```
df['index'] = df.index # 1 create new col
df = df.drop_duplicates(cols='index',
        take_last=True)# 2 use new col
del df['index']        # 3 del the col
df.sort_index(inplace=True)# 4 tidy up
```

Test if two DataFrames have same row index

```
len(a)==len(b) and all(a.index==b.index)
```

Get the integer position of a row or col index label

```
i = df.index.get_loc('row_label')
```

Trap: index.get_loc() returns an integer for a unique match. If not a unique match, may return a slice/mask.

Get integer position of rows that meet condition

```
a = np.where(df['col'] >= 2) #numpy array
```

Test if the row index values are unique/monotonic

```
if df.index.is_unique: pass # ...
b = df.index.is_monotonic_increasing
b = df.index.is_monotonic_decreasing
```

Find row index duplicates

```
if df.index.has_duplicates:
    print(df.index.duplicated())
```

Note: also similar for column label duplicates.

Working with cells

Selecting a cell by row and column labels

```
value = df.at['row', 'col']
value = df.loc['row', 'col']
value = df['col'].at['row']      # tricky
```

Note: .at[] fastest label based scalar lookup

Setting a cell by row and column labels

```
df.at['row', 'col'] = value
df.loc['row', 'col'] = value
df['col'].at['row'] = value      # tricky
```

Selecting and slicing on labels

```
df = df.loc['row1':'row3', 'col1':'col3']
```

Note: the "to" on this slice is inclusive.

Setting a cross-section by labels

```
df.loc['A':'C', 'col1':'col3'] = np.nan
df.loc[1:2,'col1':'col2']=np.zeros((2,2))
df.loc[1:2,'A':'C']=othr.loc[1:2,'A':'C']
```

Remember: inclusive "to" in the slice

Selecting a cell by integer position

```
value = df.iat[9, 3]          # [row, col]
value = df.iloc[0, 0]         # [row, col]
value = df.iloc[len(df)-1, len(df.columns)-1]
```

Selecting a range of cells by int position

```
df = df.iloc[2:4, 2:4] # subset of the df
df = df.iloc[:5, :5]   # top left corner
s = df.iloc[5, :]      # return row as Series
df = df.iloc[5:6, :]   # returns row as row
```

Note: exclusive "to" – same as python list slicing.

Setting cell by integer position

```
df.iloc[0, 0] = value         # [row, col]
df.iat[7, 8] = value
```

Setting cell range by integer position

```
df.iloc[0:3, 0:5] = value
df.iloc[1:3, 1:4] = np.ones((2, 3))
df.iloc[1:3, 1:4] = np.zeros((2, 3))
df.iloc[1:3, 1:4] = np.array([[1, 1, 1],
                              [2, 2, 2]])
```

Remember: exclusive-to in the slice

.ix for mixed label and integer position indexing

```
value = df.ix[5, 'col1']
df = df.ix[1:5, 'col1':'col3']
```

Views and copies

From the manual: Setting a copy can cause subtle errors. The rules about when a view on the data is returned are dependent on NumPy. Whenever an array of labels or a Boolean vector are involved in the indexing operation, the result will be a copy.

Summary: selecting using the DataFrame index

Using the DataFrame index to select columns

```
s =  df['col_label']         # returns Series
df = df[['col_label']]    # returns DataFrame
df = df[['L1', 'L2']] # select cols with list
df = df[index]    # select cols with an index
df = df[s]     # select with col label Series
```

Note: scalar returns Series; list &c returns a DataFrame.

Using the DataFrame index to select rows

```
df = df['from':'inc_to']  # label slice
df = df[3:7]              # integer slice
df = df[df['col'] > 0.5]  # Boolean Series
df = df.loc['label']      # single label
df = df.loc[container]    # lab list/Series
df = df.loc['from':'to']  # inclusive slice
df = df.loc[bs]           # Boolean Series
df = df.iloc[0]           # single integer
df = df.iloc[container]   # int list/Series
df = df.iloc[0:5]         # exclusive slice
df = df.ix[x]             # loc then iloc
```

Trap: Boolean Series gets rows, label Series gets cols.

Using the DataFrame index to select a cross-section

```
# r and c can be scalar, list, slice
df.loc[r, c]   # label accessor (row, col)
df.iloc[r, c]  # integer accessor
df.ix[r, c]    # label access int fallback
df[c].iloc[r]  # chained – also for .loc
```

Using the DataFrame index to select a cell

```
# r and c must be label or integer
df.at[r, c]    # fast scalar label accessor
df.iat[r, c]   # fast scalar int accessor
df[c].iat[r]   # chained – also for .at
```

DataFrame indexing methods

```
v = df.get_value(r, c)    # get by row, col
df = df.set_value(r,c,v)  # set by row, col
df = df.xs(key, axis)     # get cross-section
df = df.filter(items, like, regex, axis)
df = df.select(crit, axis)
```

Note: the indexing attributes (.loc, .iloc, .ix, .at .iat) can be used to get and set values in the DataFrame.
Note: the .loc, iloc and .ix indexing attributes can accept python slice objects. But .at and .iat do not.
Note: .loc can also accept Boolean Series arguments
Avoid: chaining in the form df[col_indexer][row_indexer]
Trap: label slices are inclusive, integer slices exclusive.

Some index attributes and methods

```
b = idx.is_monotonic_decreasing
b = idx.is_monotonic_increasing
b = idx.has_duplicates
i = idx.nlevels         # num of index levels
idx = idx.astype(dtype)# change data type
b = idx.equals(o)       # check for equality
idx = idx.union(o)      # union of two indexes
i = idx.nunique()       # number unique labels
label = idx.min()       # minimum label
label = idx.max()       # maximum label
```

Joining/Combining DataFrames

Three ways to join two DataFrames:

- merge (a database/SQL-like join operation)
- concat (stack side by side or one on top of the other)
- combine_first (splice the two together, choosing values from one over the other)

Merge on indexes

```
df_new = pd.merge(left=df1, right=df2,
          how='outer', left_index=True,
          right_index=True)
```

How: 'left', 'right', 'outer', 'inner'
How: outer=union/all; inner=intersection

Merge on columns

```
df_new = pd.merge(left=df1, right=df2,
          how='left', left_on='col1',
          right_on='col2')
```

Trap: When joining on columns, the indexes on the passed DataFrames are ignored.
Trap: many-to-many merges on a column can result in an explosion of associated data.

Join on indexes (another way of merging)

```
df_new = df1.join(other=df2, on='col1',
                  how='outer')
df_new = df1.join(other=df2,on=['a','b'],
                  how='outer')
```

Note: DataFrame.join() joins on indexes by default. DataFrame.merge() joins on common columns by default.

Simple concatenation is often the best

```
df=pd.concat([df1,df2],axis=0)#top/bottom
df = df1.append([df2, df3])   #top/bottom
df=pd.concat([df1,df2],axis=1)#left/right
```

Trap: can end up with duplicate rows or cols
Note: concat has an ignore_index parameter

Combine_first

```
df = df1.combine_first(other=df2)

# multi-combine with python reduce()
df = reduce(lambda x, y:
        x.combine_first(y),
            [df1, df2, df3, df4, df5])
```

Uses the non-null values from df1. The index of the combined DataFrame will be the union of the indexes from df1 and df2.

Groupby: Split-Apply-Combine

Grouping

```
gb = df.groupby('cat') # by one columns
gb = df.groupby(['c1','c2']) # by 2 cols
gb = df.groupby(level=0) # multi-index gb
gb = df.groupby(level=['a','b']) # mi gb
print(gb.groups)
```

Note: groupby() returns a pandas groupby object
Note: the groupby object attribute .groups contains a dictionary mapping of the groups.

Trap: NaN values in the group key are automatically dropped – there will never be a NA group.

The pandas "groupby" mechanism allows us to split the data into groups, apply a function to each group independently and then combine the results.

Iterating groups – usually not needed

```
for name, group in gb:
    print (name, group)
```

Selecting a group

```
dfa = df.groupby('cat').get_group('a')
dfb = df.groupby('cat').get_group('b')
```

Applying an aggregating function

```
# apply to a column ...
s = df.groupby('cat')['col1'].sum()
s = df.groupby('cat')['col1'].agg(np.sum)
# apply to the every column in DataFrame
s = df.groupby('cat').agg(np.sum)
df_summary = df.groupby('cat').describe()
df_row_1s = df.groupby('cat').head(1)
```

Note: aggregating functions reduce the dimension by one – they include: mean, sum, size, count, std, var, sem, describe, first, last, min, max

Applying multiple aggregating functions

```
gb = df.groupby('cat')
# apply multiple functions to one column
dfx = gb['col2'].agg([np.sum, np.mean])
# apply to multiple fns to multiple cols
dfy = gb.agg({
    'cat': np.count_nonzero,
    'col1': [np.sum, np.mean, np.std],
    'col2': [np.min, np.max]
})
```

Note: gb['col2'] above is shorthand for df.groupby('cat')['col2'], without the need for regrouping.

Transforming functions

```
# transform to group z-scores, which have
# a group mean of 0, and a std dev of 1.
zscore = lambda x: (x-x.mean())/x.std()
dfz = df.groupby('cat').transform(zscore)
# replace missing data with group mean
mean_r = lambda x: x.fillna(x.mean())
dfm = df.groupby('cat').transform(mean_r)
```

Note: can apply multiple transforming functions in a manner similar to multiple aggregating functions above,

Applying filtering functions
Filtering functions allow you to make selections based on whether each group meets specified criteria

```
# select groups with more than 10 members
eleven = lambda x: (len(x['col1']) >= 11)
df11 = df.groupby('cat').filter(eleven)
```

Group by a row index (non-hierarchical index)

```
df = df.set_index(keys='cat')
s = df.groupby(level=0)['col1'].sum()
dfg = df.groupby(level=0).sum()
```

Pivot Tables: working with long and wide data

These features work with and often create hierarchical or multi-level Indexes; (the pandas MultiIndex is powerful and complex).

Pivot, unstack, stack and melt

Pivot tables move from long format to wide format data

```
# Let's start with data in long format
from StringIO import StringIO # python2.7
#from io import StringIO      # python 3
data = """Date,Pollster,State,Party,Est
13/03/2014, Newspoll, NSW,   red,   25
13/03/2014, Newspoll, NSW,   blue,  28
13/03/2014, Newspoll, Vic,   red,   24
13/03/2014, Newspoll, Vic,   blue,  23
13/03/2014, Galaxy,   NSW,   red,   23
13/03/2014, Galaxy,   NSW,   blue,  24
13/03/2014, Galaxy,   Vic,   red,   26
13/03/2014, Galaxy,   Vic,   blue,  25
13/03/2014, Galaxy,   Qld,   red,   21
13/03/2014, Galaxy,   Qld,   blue,  27"""
df = pd.read_csv(StringIO(data),
      header=0, skipinitialspace=True)

# pivot to wide format on 'Party' column
# 1st: set up a MultiIndex for other cols
df1 = df.set_index(['Date', 'Pollster',
    'State'])
# 2nd: do the pivot
wide1 = df1.pivot(columns='Party')

# unstack to wide format on State / Party
# 1st: MultiIndex all but the Values col
df2 = df.set_index(['Date', 'Pollster',
    'State', 'Party'])
# 2nd: unstack a column to go wide on it
wide2 = df2.unstack('State')
wide3 = df2.unstack() # pop last index

# Use stack() to get back to long format
long1 = wide1.stack()
# Then use reset_index() to remove the
# MultiIndex.
long2 = long1.reset_index()

# Or melt() back to long format
# 1st: flatten the column index
wide1.columns = ['_'.join(col).strip()
    for col in wide1.columns.values]
# 2nd: remove the MultiIndex
wdf = wide1.reset_index()
# 3rd: melt away
long3 = pd.melt(wdf, value_vars=
    ['Est_blue', 'Est_red'],
    var_name='Party', id_vars=['Date',
    'Pollster', 'State'])
```

Note: See documentation, there are many arguments to these methods.

Working with dates, times and their indexes

Dates and time – points and spans

With its focus on time-series data, pandas has a suite of tools for managing dates and time: either as a point in time (a Timestamp) or as a span of time (a Period).

```
t = pd.Timestamp('2013-01-01')
t = pd.Timestamp('2013-01-01 21:15:06')
t = pd.Timestamp('2013-01-01 21:15:06.7')
p = pd.Period('2013-01-01', freq='M')
```

Note: Timestamps should be in range 1678 and 2261 years. (Check Timestamp.max and Timestamp.min).

A Series of Timestamps or Periods

```
ts = ['2015-04-01', '2014-04-02']

# Series of Timestamps (good)
s = pd.to_datetime(pd.Series(ts))

# Series of Periods (hard to make)
s = pd.Series(
    [pd.Period(x, freq='M') for x in ts] )
s = pd.Series(pd.PeriodIndex(ts,freq='D'))
```

Note: While Periods make a very useful index; they may be less useful in a Series.

From non-standard strings to Timestamps

```
t = ['09:08:55.7654-JAN092002',
        '15:42:02.6589-FEB082016']
s = pd.Series(pd.to_datetime(t,
        format="%H:%M:%S.%f-%b%d%Y"))
```

Also: %B = full month name; %m = numeric month; %y = year without century; and more …

Dates and time – stamps and spans as indexes

An index of Timestamps is a DatetimeIndex.
An index of Periods is a PeriodIndex.

```
date_strs = ['2014-01-01', '2014-04-01',
             '2014-07-01', '2014-10-01']

dti = pd.DatetimeIndex(date_strs)

pid = pd.PeriodIndex(date_strs, freq='D')
pim = pd.PeriodIndex(date_strs, freq='M')
piq = pd.PeriodIndex(date_strs, freq='Q')

print (pid[1] - pid[0])  # 90 days
print (pim[1] - pim[0])  #  3 months
print (piq[1] - piq[0])  #  1 quarter

time_strs = ['2015-01-01 02:10:40.12345',
             '2015-01-01 02:10:50.67890']
pis = pd.PeriodIndex(time_strs, freq='U')

df.index = pd.period_range('2015-01',
    periods=len(df), freq='M')

dti = pd.to_datetime(['04-01-2012'],
    dayfirst=True) # Australian date format
pi = pd.period_range('1960-01-01',
    '2015-12-31', freq='M')
```

Hint: unless you are working in less than seconds, prefer PeriodIndex over DateTimeImdex.

Period frequency constants (not a complete list)

Name	Description
U	Microsecond
L	Millisecond
S	Second
T	Minute
H	Hour
D	Calendar day
B	Business day
W-{MON, TUE, …}	Week ending on …
MS	Calendar start of month
M	Calendar end of month
QS-{JAN, FEB, …}	Quarter start with year starting (QS – December)
Q-{JAN, FEB, …}	Quarter end with year ending (Q – December)
AS-{JAN, FEB, …}	Year start (AS - December)
A-{JAN, FEB, …}	Year end (A - December)

From DatetimeIndex to Python datetime objects

```
dti = pd.DatetimeIndex(pd.date_range(
  start='1/1/2011', periods=4, freq='M'))
s = Series([1,2,3,4], index=dti)
na = dti.to_pydatetime()       # numpy array
na = s.index.to_pydatetime()   # numpy array
```

Frome Timestamps to Python dates or times

```
df['date'] = [x.date() for x in df['TS']]
df['time'] = [x.time() for x in df['TS']]
```

Note: converts to datatime.date or datetime.time. But does not convert to datetime.datetime.

From DatetimeIndex to PeriodIndex and back

```
df = DataFrame(np.random.randn(20,3))
df.index = pd.date_range('2015-01-01',
       periods=len(df), freq='M')
dfp = df.to_period(freq='M')
dft = dfp.to_timestamp()
```

Note: from period to timestamp defaults to the point in time at the start of the period.

Working with a PeriodIndex

```
pi = pd.period_range('1960-01','2015-12',
                     freq='M')
na = pi.values   # numpy array of integers
lp = pi.tolist() # python list of Periods
sp = Series(pi)  # pandas Series of Periods
ss = Series(pi).astype(str) # S of strs
ls = Series(pi).astype(str).tolist()
```

Get a range of Timestamps

```
dr = pd.date_range('2013-01-01',
    '2013-12-31', freq='D')
```

Error handling with dates

```
# 1st example returns string not Timestamp
t = pd.to_datetime('2014-02-30')
# 2nd example returns NaT (not a time)
t = pd.to_datetime('2014-02-30', coerce=True)
# NaT like NaN tests True for isnull()
b = pd.isnull(t) # --> True
```

The tail of a time-series DataFrame

```
df = df.last("5M") # the last five months
```

Upsampling and downsampling

```
# upsample from quarterly to monthly
pi = pd.period_range('1960Q1',
        periods=220, freq='Q')
df = DataFrame(np.random.rand(len(pi),5),
        index=pi)
dfm = df.resample('M', convention='end')
# use ffill or bfill to fill with values

# downsample from monthly to quarterly
dfq = dfm.resample('Q', how='sum')
```

Time zones

```
t = ['2015-06-30 00:00:00',
      '2015-12-31 00:00:00']
dti = pd.to_datetime(t
      ).tz_localize('Australia/Canberra')
dti = dti.tz_convert('UTC')
ts = pd.Timestamp('now',
      tz='Europe/London')

# get a list of all time zones
import pyzt
for tz in pytz.all_timezones:
    print tz
```

Note: by default, Timestamps are created without time zone information.

Row selection with a time-series index

```
# start with the play data above
idx = pd.period_range('2015-01',
        periods=len(df), freq='M')
df.index = idx

february_selector = (df.index.month == 2)
february_data = df[february_selector]

q1_data = df[(df.index.month >= 1) &
   (df.index.month <= 3)]

mayornov_data = df[(df.index.month == 5) |
   (df.index.month == 11)]

totals = df.groupby(df.index.year).sum()
```

Also: year, month, day [of month], hour, minute, second, dayofweek [Mon=0 .. Sun=6], weekofmonth, weekofyear [numbered from 1], week starts on Monday], dayofyear [from 1], …

The Series.dt accessor attribute

DataFrame columns that contain datetime-like objects can be manipulated with the .dt accessor attribute

```
t = ['2012-04-14 04:06:56.307000',
     '2011-05-14 06:14:24.457000',
     '2010-06-14 08:23:07.520000']

# a Series of time stamps
s = pd.Series(pd.to_datetime(t))
print(s.dtype)      # datetime64[ns]
print(s.dt.second)  # 56, 24,  7
print(s.dt.month)   #  4,  5,  6
# a Series of time periods
s = pd.Series(pd.PeriodIndex(t,freq='Q'))
print(s.dtype)      # datetime64[ns]
print(s.dt.quarter) #  2,  2,  2
print(s.dt.year)    #  2012, 2011, 2010
```

Plotting from the DataFrame

Import matplotlib, choose a matplotlib style

```
import matplotlib.pyplot as plt
print(plt.style.available)
plt.style.use('ggplot')
```

Fake up some data (which we reuse repeatedly)

```
a = np.random.normal(0,1,999)
b = np.random.normal(1,2,999)
c = np.random.normal(2,3,999)
df = pd.DataFrame([a,b,c]).T
df.columns =['A', 'B', 'C']
```

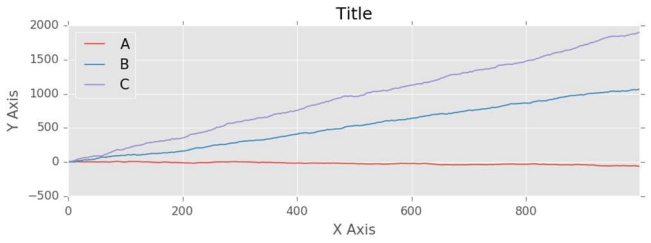
Line plot

```
df1 = df.cumsum()
ax = df1.plot()

# from here down - standard plot output
ax.set_title('Title')
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')

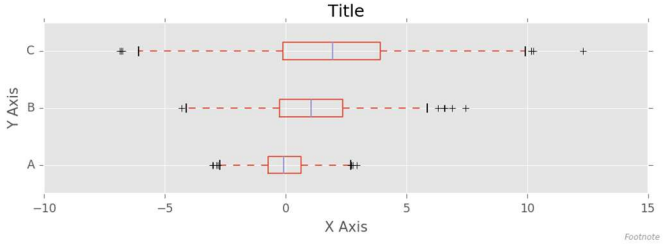
fig = ax.figure
fig.set_size_inches(8, 3)
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)

plt.close()
```



Box plot

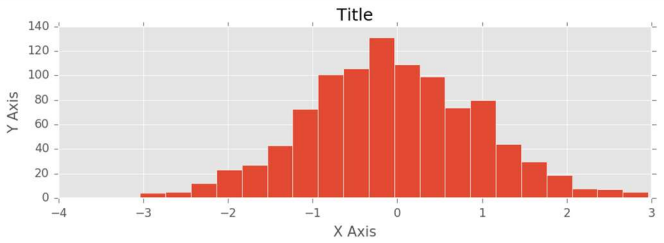
```
ax = df.plot.box(vert=False)
# followed by the standard plot code as above
```




```
ax = df.plot.box(column='c1', by='c2')
```

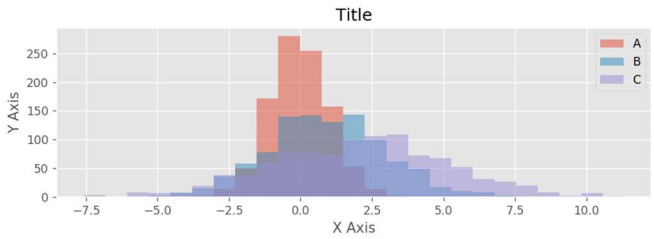
Histogram

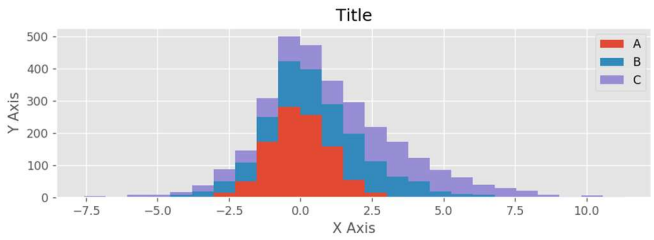
```
ax = df['A'].plot.hist(bins=20)
# followed by the standard plot code as above
```



Multiple histograms (overlapping or stacked)

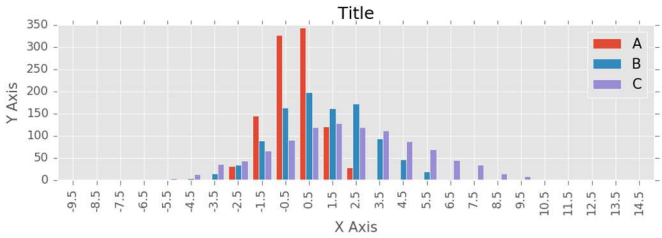
```
ax = df.plot.hist(bins=25, alpha=0.5) # or...
ax = df.plot.hist(bins=25, stacked=True)
# followed by the standard plot code as above
```





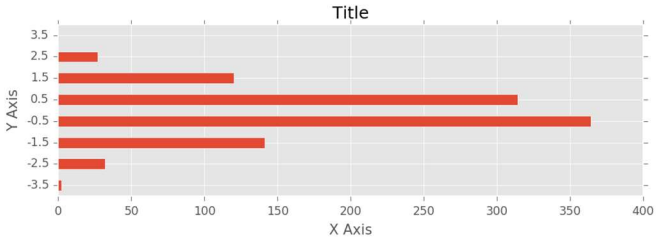
Bar plots

```
bins = np.linspace(-10,15,26)
binned = pd.DataFrame()
for x in df.columns:
    y=pd.cut(df[x],bins,labels=bins[:-1])
    y=y.value_counts().sort_index()
    binned = pd.concat([binned,y],axis=1)
binned.index = binned.index.astype(float)
binned.index += (np.diff(bins) / 2.0)
ax = binned.plot.bar(stacked=False,
    width=0.8) # for bar width
# followed by the standard plot code as above
```



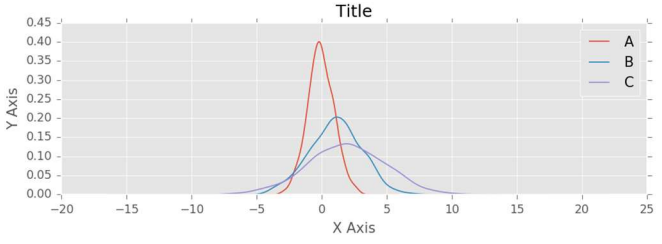
Horizontal bars

```
ax = binned['A'][(binned.index >= -4) &
    (binned.index <= 4)].plot.barh()
# followed by the standard plot code as above
```



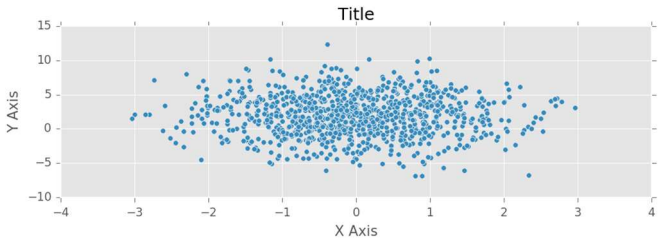
Density plot

```
ax = df.plot.kde()
# followed by the standard plot code as above
```



Scatter plot

```
ax = df.plot.scatter(x='A', y='C')
# followed by the standard plot code as above
```



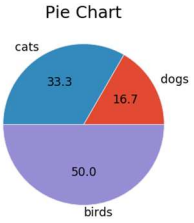
Pie chart

```
s = pd.Series(data=[10, 20, 30],
    index = ['dogs', 'cats', 'birds'])
ax = s.plot.pie(autopct='%.1f')

# followed by the standard plot output ...
ax.set_title('Pie Chart')
ax.set_aspect(1) # make it round
ax.set_ylabel('') # remove default

fig = ax.figure
fig.set_size_inches(8, 3)
fig.savefig('filename.png', dpi=125)

plt.close(fig)
```



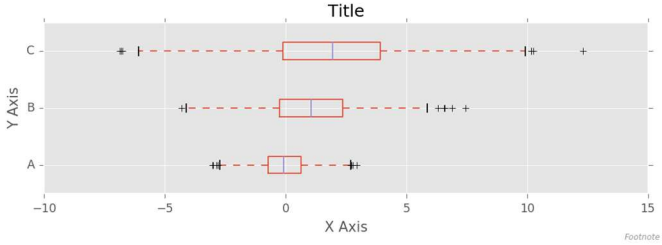
Change the range plotted

```
ax.set_xlim([-5, 5])

# for some white space on the chart ...
lower, upper = ax.get_ylim()
ax.set_ylim([lower-1, upper+1])
```

Add a footnote to the chart

```
# after the fig.tight_layout(pad=1) above
fig.text(0.99, 0.01, 'Footnote',
    ha='right', va='bottom',
    fontsize='x-small',
    fontstyle='italic', color='#999999')
```



A line and bar on the same chart

In matplotlib, bar charts visualise categorical or discrete data. Line charts visualise continuous data. This makes it hard to get bars and lines on the same chart. Typically combined charts either have too many labels, and/or the lines and bars are misaligned or missing. You need to trick matplotlib a bit … pandas makes this tricking easier

```
# start with fake percentage growth data
s = pd.Series(np.random.normal(
    1.02, 0.015, 40))
s = s.cumprod()
dfg = (pd.concat([s / s.shift(1),
    s / s.shift(4)], axis=1) * 100) - 100
dfg.columns = ['Quarter', 'Annual']
dfg.index = pd.period_range('2010-Q1',
    periods=len(dfg), freq='Q')

# reindex with integers from 0; keep old
old = dfg.index
dfg.index = range(len(dfg))

# plot the line from pandas
ax = dfg['Annual'].plot(color='blue',
    label='Year/Year Growth')

# plot the bars from pandas
dfg['Quarter'].plot.bar(ax=ax,
    label='Q/Q Growth', width=0.8)

# relabel the x-axis more appropriately
ticks = dfg.index[((dfg.index+0)%4)==0]
labs = pd.Series(old[ticks]).astype(str)
ax.set_xticks(ticks)
ax.set_xticklabels(labs.str.replace('Q',
    '\nQ'), rotation=0)

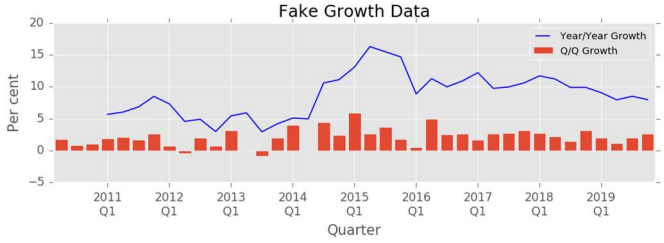
# fix the range of the x-axis … skip 1st
ax.set_xlim([0.5,len(dfg)-0.5])

# add the legend
l=ax.legend(loc='best',fontsize='small')

# finish off and plot in the usual manner
ax.set_title('Fake Growth Data')
ax.set_xlabel('Quarter')
ax.set_ylabel('Per cent')

fig = ax.figure
fig.set_size_inches(8, 3)
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)

plt.close()
```



Working with missing and non-finite data

Working with missing data

Pandas uses the not-a-number construct (np.nan and float('nan')) to indicate missing data. The Python None can arise in data as well. It is also treated as missing data; as is the pandas not-a-time construct (pandas.NaT).

Missing data in a Series

```
s = Series( [8,None,float('nan'),np.nan])
          #[8,     NaN,   NaN,   NaN]
s.isnull() #[False, True,  True,  True]
s.notnull()#[True,  False, False, False]
s.fillna(0)#[8,     0,     0,     0]
```

Missing data in a DataFrame

```
df = df.dropna() # drop all rows with NaN
df = df.dropna(axis=1) # same for cols
df=df.dropna(how='all') #drop all NaN row
df=df.dropna(thresh=2) # drop 2+ NaN in r
# only drop row if NaN in a specified col
df = df.dropna(df['col'].notnull())
```

Recoding missing data

```
df.fillna(0, inplace=True) # np.nan → 0
s = df['col'].fillna(0)    # np.nan → 0
df = df.replace(r'\s+', np.nan,
      regex=True) # white space → np.nan
```

Non-finite numbers

With floating point numbers, pandas provides for positive and negative infinity.

```
s = Series([float('inf'), float('-inf'),
     np.inf, -np.inf])
```

Pandas treats integer comparisons with plus or minus infinity as expected.

Testing for finite numbers

(using the data from the previous example)

```
b = np.isfinite(s)
```

Working with Categorical Data

Categorical data

The pandas Series has an R factors-like data type for encoding categorical data.

```
s = Series(['a','b','a','c','b','d','a'],
                        dtype='category')
df['B'] = df['A'].astype('category')
```

Note: the key here is to specify the "category" data type.
Note: categories will be ordered on creation if they are sortable. This can be turned off. See ordering below.

Convert back to the original data type

```
s = Series(['a','b','a','c','b','d','a'],
                        dtype='category')
s = s.astype('string')
```

Ordering, reordering and sorting

```
s = Series(list('abc'), dtype='category')
print (s.cat.ordered)
s=s.cat.reorder_categories(['b','c','a'])
s = s.sort()
s.cat.ordered = False
```

Trap: category must be ordered for it to be sorted

Renaming categories

```
s = Series(list('abc'), dtype='category')
s.cat.categories = [1, 2, 3] # in place
s = s.cat.rename_categories([4,5,6])
# using a comprehension ...
s.cat.categories = ['Group ' + str(i)
    for i in s.cat.categories]
```

Trap: categories must be uniquely named

Adding new categories

```
s = s.cat.add_categories([4])
```

Removing categories

```
s = s.cat.remove_categories([4])
s.cat.remove_unused_categories() #inplace
```

Working with strings

Working with strings

```
# assume that df['col'] is series of strings
s = df['col'].str.lower()
s = df['col'].str.upper()
s = df['col'].str.len()

# the next set work like Python
df['col'] += 'suffix'       # append
df['col'] *= 2              # duplicate
s = df['col1'] + df['col2'] # concatenate
```

Most python string functions are replicated in the pandas DataFrame and Series objects.

Regular expressions

```
s = df['col'].str.contains('regex')
s = df['col'].str.startswith('regex')
s = df['col'].str.endswith('regex')
s = df['col'].str.replace('old', 'new')
df['b'] = df.a.str.extract('(pattern)')
```

Note: pandas has many more regex methods.

Basic Statistics

Summary statistics

```
s = df['col1'].describe()
df1 = df.describe()
```

DataFrame – key stats methods

```
df.corr()     # pairwise correlation cols
df.cov()      # pairwise covariance cols
df.kurt()     # kurtosis over cols (def)
df.mad()      # mean absolute deviation
df.sem()      # standard error of mean
df.var()      # variance over cols (def)
```

Value counts

```
s = df['col1'].value_counts()
```

Cross-tabulation (frequency count)

```
ct = pd.crosstab(index=df['a'],
        cols=df['b'])
```

Quantiles and ranking

```
quants = [0.05, 0.25, 0.5, 0.75, 0.95]
q = df.quantile(quants)
r = df.rank()
```

Histogram binning

```
count, bins = np.histogram(df['col1'])
count, bins = np.histogram(df['col'],
              bins=5)
count, bins = np.histogram(df['col1'],
              bins=[-3,-2,-1,0,1,2,3,4])
```

Regression

```
import statsmodels.formula.api as sm
result = sm.ols(formula="col1 ~ col2 +
          col3", data=df).fit()
print (result.params)
print (result.summary())
```

Simple smoothing example using a rolling apply

```
k3x5 = np.array([1,2,3,3,3,2,1]) / 15.0
s = df['A'].rolling(window=len(k3x5),
    min_periods=len(k3x5),
    center=True).apply(
         func=lambda x: (x * k3x5).sum())
# fix the missing end data ... unsmoothed
s = df['A'].where(s.isnull(), other=s)
```

Cautionary note

This cheat sheet was cobbled together by tireless bots roaming the dark recesses of the Internet seeking ursine and anguine myths from a fabled land of milk and honey where it is rumoured pandas and pythons gambol together. There is no guarantee the narratives were captured and transcribed accurately. You use these notes at your own risk. You have been warned. I will not be held responsible for whatever happens to you and those you love once your eyes begin to see what is written here.

Version: This cheat sheet was last updated with Python 3.6 and pandas 0.19.2 in mind.

Errors: If you find any errors, please email me at markthegraph@gmail.com; (but please do not correct my use of Australian-English spelling conventions).

Python OOP Basics

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or properties), and the code is in the form of procedures (often known as methods).

Encapsulation

Encapsulation is one of the fundamental concepts of object-oriented programming, which helps to protect the data and methods of an object from unauthorized access and modification. It is a way to achieve data abstraction, which means that the implementation details of an object are hidden from the outside world, and only the essential information is exposed.

In Python, encapsulation can be achieved by using access modifiers. Access modifiers are keywords that define the accessibility of attributes and methods in a class. The three access modifiers available in Python are public, private, and protected. However, Python does not have an explicit way of defining access modifiers like some other programming languages such as Java and C++. Instead, it uses a convention of using underscore prefixes to indicate the access level.

In the given code example, the class MyClass has two attributes, _protected_var and __private_var. The _protected_var is marked as protected by using a single underscore prefix. This means that the attribute can be accessed within the class and its subclasses but not outside the class. The __private_var is marked as private by using two underscore prefixes. This means that the attribute can only be accessed within the class and not outside the class, not even in its subclasses.

When we create an object of the MyClass class, we can access the _protected_var attribute using the object name with a single underscore prefix. However, we cannot access the __private_var attribute using the object name, as it is hidden from the outside world. If we try to access the __private_var attribute, we will get an AttributeError as shown in the code.

In summary, encapsulation is an important concept in object-oriented programming that helps to protect the implementation details of an object. In Python, we can achieve encapsulation by using access modifiers and using underscore prefixes to indicate the access level.

```
# Define a class named MyClass
class MyClass:

    # Constructor method that initializes the class object
    def __init__(self):

        # Define a protected variable with an initial value of 10
        # The variable name starts with a single underscore, which in
        self._protected_var = 10

        # Define a private variable with an initial value of 20
        # The variable name starts with two underscores, which indica
        self.__private_var = 20

# Create an object of MyClass class
obj = MyClass()

# Access the protected variable using the object name and print its v
# The protected variable can be accessed outside the class but
# it is intended to be used within the class or its subclasses
print(obj._protected_var)   # output: 10

# Try to access the private variable using the object name and print
# The private variable cannot be accessed outside the class, even by
# This will raise an AttributeError because the variable is not acces
print(obj.__private_var)    # AttributeError: 'MyClass' object has no
```

Inheritance

Inheritance promotes code reuse and allows you to create a hierarchy of classes that share common attributes and methods. It helps in creating clean and organized code by keeping related functionality in one place and promoting the concept of modularity. The base class from which a new class is derived is also known as a parent class, and the new class is known as the child class or subclass.

In the code, we define a class named Animal which has a constructor method that initializes the class object with a name attribute and a method named speak. The speak method is defined in the Animal class but does not have a body.

We then define two subclasses named Dog and Cat which inherit from the Animal class. These subclasses override the speak method of the Animal class.

We create a Dog object with a name attribute “Rover” and a Cat object with a name attribute “Whiskers”. We call the speak method of the Dog object using dog.speak(), and it prints “Woof!” because the speak method of the Dog class overrides the speak method of the Animal class. Similarly, we call the speak method of the Cat object using cat.speak(), and it prints “Meow!” because the speak method of the Cat class overrides the speak method of the Animal class.

```
# Define a class named Animal
class Animal:

    # Constructor method that initializes the class object with a nam
    def __init__(self, name):
        self.name = name

    # Method that is defined in the Animal class but does not have a
    # This method will be overridden in the subclasses of Animal
    def speak(self):
        print("")

# Define a subclass named Dog that inherits from the Animal class
class Dog(Animal):

    # Override the speak method of the Animal class
    def speak(self):
        print("Woof!")

# Define a subclass named Cat that inherits from the Animal class
```

```
class Cat(Animal):

    # Override the speak method of the Animal class
    def speak(self):
        print("Meow!")

# Create a Dog object with a name attribute "Rover"
dog = Dog("Rover")

# Create a Cat object with a name attribute "Whiskers"
cat = Cat("Whiskers")

# Call the speak method of the Dog class and print the output
# The speak method of the Dog class overrides the speak method of the
# Therefore, when we call the speak method of the Dog object, it will
dog.speak()   # output: Woof!

# Call the speak method of the Cat class and print the output
# The speak method of the Cat class overrides the speak method of the
# Therefore, when we call the speak method of the Cat object, it will
cat.speak()   # output: Meow!
```

Polymorphism

Polymorphism is an important concept in object-oriented programming that allows you to write code that can work with objects of different classes in a uniform way. In Python, polymorphism is achieved by using method overriding or method overloading.

Method overriding is when a subclass provides its own implementation of a method that is already defined in its parent class. This allows the subclass to modify the behavior of the method without changing its name or signature.

Method overloading is when multiple methods have the same name but different parameters. Python does not support method overloading directly, but it can be achieved using default arguments or variable-length arguments.

Polymorphism makes it easier to write flexible and reusable code. It allows you to write code that can work with different objects without needing to know their specific types.

```python
#The Shape class is defined with an abstract area method, which is in
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    # The Rectangle class is defined with an __init__ method that ini
    # width and height instance variables.
    # It also defines an area method that calculates and returns
    # the area of a rectangle using the width and height instance var
    def __init__(self, width, height):
        self.width = width  # Initialize width instance variable
        self.height = height  # Initialize height instance variable

    def area(self):
        return self.width * self.height  # Return area of rectangle


 # The Circle class is defined with an __init__ method
 # that initializes a radius instance variable.
 # It also defines an area method that calculates and
 # returns the area of a circle using the radius instance variable.
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius  # Initialize radius instance variable

    def area(self):
        return 3.14 * self.radius ** 2  # Return area of circle using

# The shapes list is created with one Rectangle object and one Circle
# loop iterates over each object in the list and calls the area metho
# The output will be the area of the rectangle (20) and the area of t
shapes = [Rectangle(4, 5), Circle(7)]  # Create a list of Shape objec
for shape in shapes:
    print(shape.area())  # Output the area of each Shape object
```

Abstraction

Abstraction is an important concept in object-oriented programming (OOP) because it allows you to focus on the essential features of an object or system while ignoring the details that aren't relevant to the current context. By reducing complexity and hiding unnecessary details, abstraction can make code more modular, easier to read, and easier to maintain.

In Python, abstraction can be achieved by using abstract classes or interfaces. An abstract class is a class that cannot be instantiated directly, but is meant to be subclassed by other classes. It often includes abstract methods that have no implementation, but provide a template for how the subclass should be implemented. This allows the programmer to define a common interface for a group of related classes, while still allowing each class to have its own specific behavior.

An interface, on the other hand, is a collection of method signatures that a class must implement in order to be considered "compatible" with the interface. Interfaces are often used to define a common set of methods that multiple classes can implement, allowing them to be used interchangeably in certain contexts.

Python does not have built-in support for abstract classes or interfaces, but they can be implemented using the abc (abstract base class) module. This module provides the ABC class and the abstractmethod decorator, which can be used to define abstract classes and methods.

Overall, abstraction is a powerful tool for managing complexity and improving code quality in object-oriented programming, and Python provides a range of options for achieving abstraction in your code.

```
# Import the abc module to define abstract classes and methods
from abc import ABC, abstractmethod
```



```
        pass

# Define a Rectangle class that inherits from Shape
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    # Implement the area method for Rectangles
    def area(self):
        return self.width * self.height

# Define a Circle class that also inherits from Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Implement the area method for Circles
    def area(self):
        return 3.14 * self.radius ** 2

# Create a list of shapes that includes both Rectangles and Circles
shapes = [Rectangle(4, 5), Circle(7)]

# Loop through each shape in the list and print its area
for shape in shapes:
    print(shape.area())
```

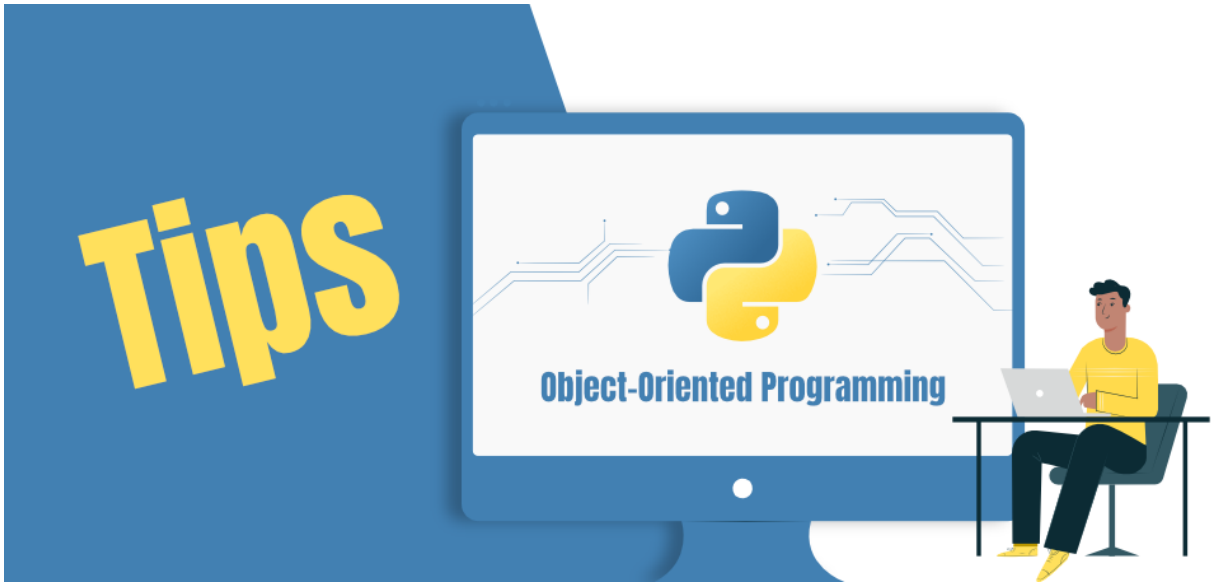
These are some of the basic OOP principles in Python. This page is currently in progress and more detailed examples and explanations will be coming soon.

Previous page
Context manager

Next page
Dataclasses

8 Tips For Object-Oriented Programming in Python

OOP or Object-Oriented Programming is a programming paradigm that organizes software design around data or objects and relies on the concept of classes and objects, rather than functions and logic. **Object-oriented programming ensures code reusability and prevents Redundancy**, and hence has become very popular even in the fields outside software engineering like **Machine Learning, Artificial Intelligence, Data Science**, etc. There are many object-oriented programming languages like Java, JavaScript, C++, Python, and many others.



Basics of Object-Oriented Programming in Python

As stated earlier, **OOP is a programming paradigm that uses objects and classes and aims to implement real-world entities in programming.** The main concepts of OOP are stated below:

1. Inheritance: It is the process in which **one class inherits the methods and attributes from another class**. The class whose properties and methods are inherited is known as the Parent class and the class which inherits from the parent class is called the Child class. Inheritance is the most important aspect of object-oriented programming and provides the reusability of the code.

2. Encapsulation: The word, “encapsulate” means to enclose something and the principle of encapsulation works in the same way. In OOP, **the data and functions which operate on that data are wrapped together into a single unit by encapsulation.** By this method, we can hide private details of a class and can only expose the functionality that is needed for interfacing with it.

3. Polymorphism: Polymorphism is taken from the Greek words Poly and morph which means many and shape respectively. Hence in OOP, **Polymorphism refers to the functions having the same names but having different functionalities.** Polymorphism helps in making programming more intuitive and easier.

4. Data Abstraction: Abstraction is another functionality of OOP in which **we hide the internal details or implementations of a function and show the functionalities only**. In other words, the users know “what the function does” but they don’t know “how it does” because they only get to see the basic implementation of the function, whereas the inner working is hidden.

To know more about the 4 pillars of Object-oriented programming click here. In this blog, we will discuss the **8 Tips for Object-Oriented Programming in Python**. So Let’s take a look.

1. Difference of Class and Instance Level Data

To under inheritance, **learning to differentiate between class-level and instance-level data is very crucial**, as inheritance is one of the core concepts of OOP. An instance attribute is a Python variable belonging to one object and is unique to it. It is accessible only in the scope of this object and is defined inside the constructor function of the class. Whereas, a class attribute is unique to a class rather than a particular object and is defined outside the constructor function. While writing a code, Instance-level data should be kept separate from class-level data and should not interfere with it.

2. Using Meaningful Names

One of the best practices to follow while writing Object-Oriented Programming is to use meaningful names keeping in mind that the **naming convention must follow camel casing.** There are rules which apply while naming attributes and functions as well. Hence, we should always name the design in such a way that one class, attribute, or function is responsible for one particular task only.

3. Knowing the Use of Static

Static means that the member is bound to a class level rather than to an instance level. A static method can not access or modify the state of the class. By using static, the method doesn't need access to the class instance. Also, using **static methods improves code readability and saves a lot of memory**.

4. Deciding Between Internal and Private Modifiers

Access specifiers in Python play an important role in securing data from

a private one a double leading underscore __ is used. This practice helps in determining the access control of a specific data member or a member function of a class and tells the users which parts of the code are not public and should not be relied on.

5. Docstrings in Python

Developers are used to writing comments in the code but comments do not always provide the most structured way of workflows. For this inconvenience, **Python structured documentation or docstrings provide a convenient way of associating documentation with Python public modules, functions, classes, methods**, etc. Unlike source code comments, the docstring describes what the function does and not how. Docstrings in python are declared using "'triple single quotes"' or """triple double quotes""".

6. pep8 Guidelines

Python Enterprise Proposal also called **PEP** are coding conventions for python packages. Besides writing a code with proper logic, many other important factors

affect a code's quality. PEP 8 is a document that was written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan with its main aim to enhance the readability and consistency of code. One should keep in mind three important pep8 guidelines that are, to ensure that **each line of code is limited to 80 characters**, **all libraries should be imported at the beginning, and to eliminate redundant variables or intermediary variables present in the code**.

7. Setting Access to Attributes

Attributes of a class are function objects that are used to implement access controls of the classes and define corresponding methods of its instances. To access and manipulate the attributes of a class, python has some in-built methods. Those are **getattr()**, **hasattr()**, **setattr()** and **delattr()**. The **getattr()** function helps in accessing the attribute of the object. The **hasattr()** function is used in checking if an attribute exists or not. **Setattr()** is used to set an attribute. In case the attribute does not exist, then one would be created, whereas, **delattr()** is used to delete an attribute.

8. Using Abstract Classes

An abstract class provides a **common interface for different implementations of a component**. In object-oriented programming, developers need to reduce the amount of coding, and hence, abstract classes are used. An abstract class is used as it applies to a wide variety of objects and helps in the creation of a set of methods that must be created within any child classes built from that abstract class.

Last Updated : 08 Mar, 2022

Previous

Top 10 Python Applications in Real World

Next

10 Skills to Become a Backend Developer in 2023

Similar Reads

- Object Oriented Programming in Python | Set 2 (Data Hiding and Object Printing)
- A Step-by-Step Approach to Learn Object Oriented Programming
- Top 10 Object-Oriented Programming Languages in 2023
- Python | Matplotlib Sub plotting using object oriented API
- Python | Matplotlib Graph plotting using object oriented API
- Object oriented testing in Python
- ChatGPT Tips & Tricks: 5 Secret Tips That Can Boost Your ChatGPT Game to God Mode
- Characteristics of Good Object Oriented Design
- 6 Steps To Approach Object-Oriented Design Questions in Interview
- Tips to reduce Python object size

Complete Tutorials

- OpenAI Python API - Complete Guide
- ChatGPT Tutorial: ChatGPT-3.5 Guide for Beginners
- Pandas AI: The Generative AI Python Library
- Python for Kids - Fun Tutorial to Learn Python Programming
- Data Analysis Tutorial

Article Contributed By :

priyankab14

priyankab14

Follow

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Article Tags : GBlog , Python

Practice Tags : python

Improve Article | Report Issue

GeeksforGeeks
Sanchhaya Education Private Limited
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

Apply for Mentor

GeeksforGeeks Videos

All Cheat Sheets

NumPy Tutorial

Deep Learning Tutorial

Computer Science

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Python

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

OpenCV Python Tutorial

Python Interview Question

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP

DevOps Roadmap

Competitive Programming

Top DS or Algo for CP

Top 50 Tree

Top 50 Graph

Top 50 Array

Top 50 String

Top 50 DP

Top 15 Websites for CP

System Design

What is System Design

Monolithic and Distributed SD

High Level Design or HLD

Low Level Design or LLD

Crack System Design Round

System Design Interview Questions

Grokking Modern System Design

JavaScript

TypeScript

ReactJS

NextJS

AngularJS

NodeJS

Express.js

Lodash

Web Browser

TextBook Solutions/Study Material

NCERT Solutions for Class 12

NCERT Solution for Class 11

NCERT Solutions for Class 10

NCERT Solutions for Class 9

NCERT Solutions for Class 8

Complete Study Material

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

Commerce

Accountancy

Business Studies

Indian Economics

Macroeconomics

Microeconimics

Statistics for Economics

Management & Finance

Management

HR Managament

Income Tax

Finance

Economics

SSC/ BANKING

SSC CGL Syllabus

SBI PO Syllabus

SBI Clerk Syllabus

IBPS PO Syllabus

IBPS Clerk Syllabus

SSC CGL Practice Papers

Colleges

Indian Colleges Admission & Campus Experiences

Top Engineering Colleges

Top BCA Colleges

Top MBA Colleges

Top Architecture College

Choose College For Graduation

Companies

Preparation Corner

Exams

More Tutorials

Write & Earn

Introduction of Object Oriented Programming

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts:

- Class
- Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

1. Class:

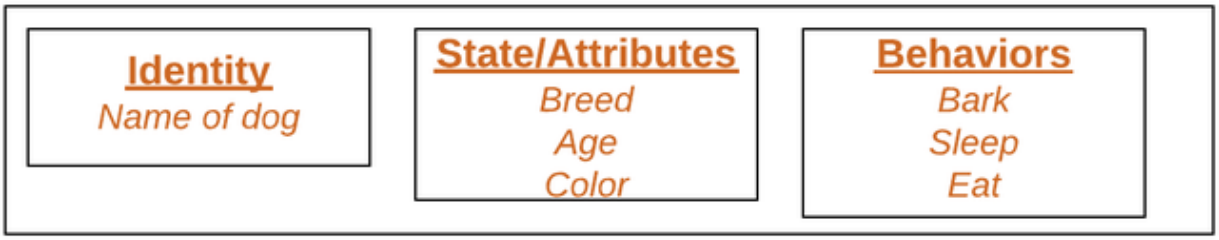
A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

2. Object:

It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

For example "Dog" is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.



Object

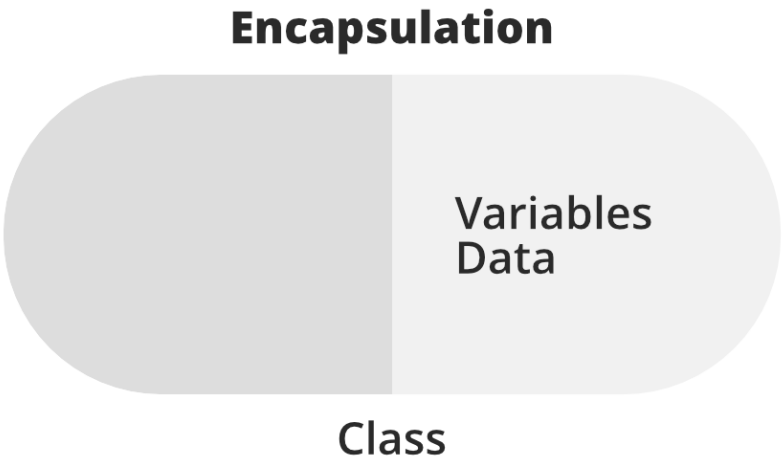
3. Data Abstraction:

Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

4. Encapsulation:

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are

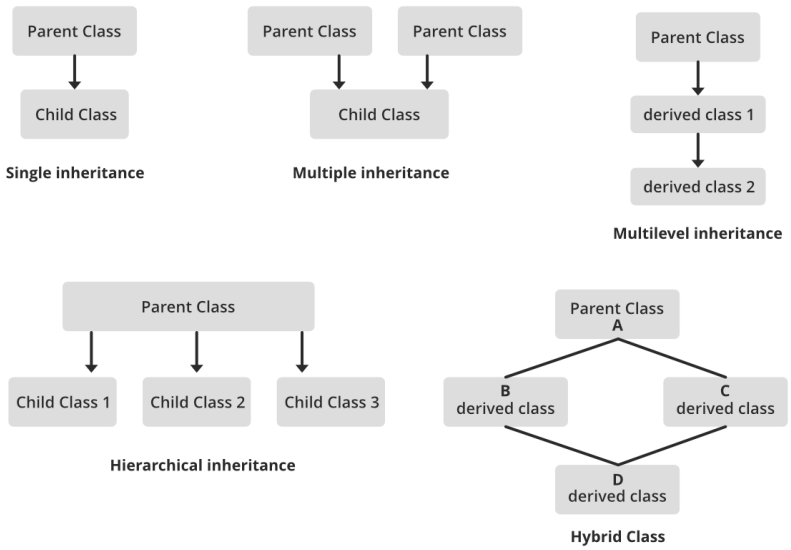
declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.



Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

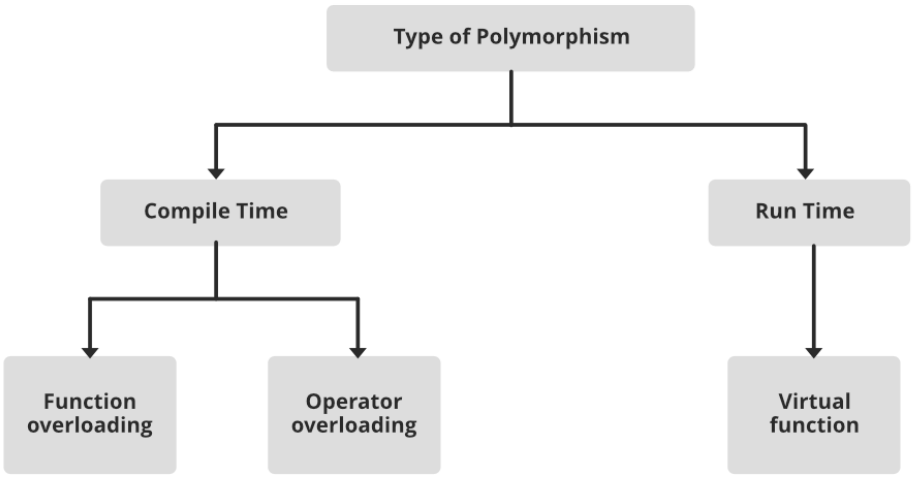
5. Inheritance:

Inheritance is an important pillar of OOP(Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.



6. Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.



7. Dynamic Binding:

In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. Dynamic Method Binding One of the main advantages of inheritance is that some derived class D has all the members of its base class B. Once D is not hiding any of the public members of B, then an object of D can represent B in any context where a B could be used. This feature is known as subtype polymorphism.

8. Message Passing:

It is a form of communication used in object-oriented programming as well as parallel programming. Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for

Why do we need object-oriented programming

- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code: which will work with a range of data, so we don't have to write basic stuff over and over again.

Last Updated : 09 Feb, 2023

Previous
BCA 3rd Semester Syllabus (2023)

Next
Abstraction in C++

Similar Reads

Object Oriented Programming in Python | Set 2 (Data Hiding and Object Printing)

Difference between Functional Programming and Object Oriented Programming

Differences between Procedural and Object Oriented Programming

Object-Oriented Programming in Ruby | Set 1

Object Oriented Programming in Ruby | Set-2

Object Oriented Programming (OOPs) in MATLAB

Design Goals and Principles of Object Oriented Programming

Top 10 Object-Oriented Programming Languages in 2023

What is COBOL(Common Business Oriented Language)?

Introduction to Swift Programming

Article Contributed By :

sambhav228

sambhav228

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Improved By : maulyashetty710

Article Tags : Programming Language , School Programming

Improve Article | Report Issue

GeeksforGeeks
Sanchhaya Education Private Limited
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

Company

- About Us
- Legal
- Careers
- In Media
- Contact Us
- Advertise with us
- GFG Corporate Solution
- Placement Training Program
- Apply for Mentor

Explore

- Job-A-Thon Hiring Challenge
- Hack-A-Thon
- GfG Weekly Contest
- Offline Classes (Delhi/NCR)
- DSA in JAVA/C++
- Master System Design
- Master CP
- GeeksforGeeks Videos

Languages

- Python
- Java
- C++
- PHP
- GoLang
- SQL
- R Language
- Android Tutorial

DSA

- Data Structures
- Algorithms
- DSA for Beginners
- Basic DSA Problems
- DSA Roadmap
- Top 100 DSA Interview Problems
- DSA Roadmap by Sandeep Jain
- All Cheat Sheets

Data Science & ML

- Data Science With Python
- Data Science For Beginner
- Machine Learning Tutorial
- Maths For Machine Learning
- Pandas Tutorial
- NumPy Tutorial
- Deep Learning Tutorial

HTML & CSS

- HTML
- CSS
- Bootstrap
- Tailwind CSS
- SASS
- LESS
- NodeJS
- Web Design

Computer Science

- GATE CS Notes
- Operating Systems
- Computer Network
- Database Management System
- Software Engineering
- Digital Logic Design
- Engineering Maths

Python

- Python Programming Examples
- Django Tutorial
- Python Projects
- Python Tkinter
- OpenCV Python Tutorial
- Python Interview Question

DevOps

- Git
- AWS
- Docker
- Kubernetes
- Azure
- GCP
- DevOps Roadmap

Competitive Programming

- Top DS or Algo for CP
- Top 50 Tree
- Top 50 Graph
- Top 50 Array
- Top 50 String
- Top 50 DP
- Top 15 Websites for CP

System Design

- What is System Design
- Monolithic and Distributed SD
- High Level Design or HLD
- Low Level Design or LLD
- Crack System Design Round
- System Design Interview Questions
- Grokking Modern System Design

JavaScript

- TypeScript
- ReactJS
- NextJS
- AngularJS
- NodeJS
- Express.js
- Lodash
- Web Browser

TextBook Solutions/Study

School Subjects

Commerce

- Accountancy

Management & Finance

SSC/ BANKING

Colleges

Material

- NCERT Solutions for Class 12
- NCERT Solution for Class 11
- NCERT Solutions for Class 10
- NCERT Solutions for Class 9
- NCERT Solutions for Class 8
- Complete Study Material
- Mathematics
- Physics
- Chemistry
- Biology
- Social Science
- English Grammar
- Business Studies
- Indian Economics
- Macroeconomics
- Microeconimics
- Statistics for Economics
- Management
- HR Managament
- Income Tax
- Finance
- Economics
- SSC CGL Syllabus
- SBI PO Syllabus
- SBI Clerk Syllabus
- IBPS PO Syllabus
- IBPS Clerk Syllabus
- SSC CGL Practice Papers
- Indian Colleges Admission & Campus Experiences
- Top Engineering Colleges
- Top BCA Colleges
- Top MBA Colleges
- Top Architecture College
- Choose College For Graduation

Companies

- IT Companies
- Software Development Companies
- Artificial Intelligence(AI) Companies
- CyberSecurity Companies
- Service Based Companies
- Product Based Companies
- PSUs for CS Engineers

Preparation Corner

- Company Wise Preparation
- Preparation for SDE
- Experienced Interviews
- Internship Interviews
- Competitive Programming
- Aptitude Preparation
- Puzzles

Exams

- JEE Mains
- JEE Advanced
- GATE CS
- NEET
- UGC NET
- CAT

More Tutorials

- Software Testing
- Software Development
- Product Management
- SAP
- SEO
- Linux
- Excel

Write & Earn

- Write an Article
- Improve an Article
- Pick Topics to Write
- Share your Experiences
- Internships

Object Oriented Programming in Python | Set 2 (Data Hiding and Object Printing)

Prerequisite: Object-Oriented Programming in Python | Set 1 (Class, Object and Members)

Data hiding

In Python, we use double underscore (Or __) before the attributes name and those attributes will not be directly visible outside.

Python

```
class MyClass:

    # Hidden member of MyClass
    __hiddenVariable = 0
```



```
            print (self.__hiddenVariable)

# Driver code
myObject = MyClass()
myObject.add(2)
myObject.add(5)

# This line causes error
print (myObject.__hiddenVariable)
```

Output :

```
2
7
Traceback (most recent call last):
  File "filename.py", line 13, in
    print (myObject.__hiddenVariable)
AttributeError: MyClass instance has
no attribute '__hiddenVariable'
```

In the above program, we tried to access a hidden variable outside the class using an object and it threw an exception.
We can access the value of a hidden attribute by a tricky syntax:

Python


```python
# A Python program to demonstrate that hidden
# members can be accessed outside a class
class MyClass:

    # Hidden member of MyClass
    __hiddenVariable = 10

# Driver code
myObject = MyClass()
print(myObject._MyClass__hiddenVariable)
```

Output :

```
10
```

Private methods are accessible outside their class, just not easily accessible.
Nothing in Python is truly private; internally, the names of private methods and

attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names [See this for source].

Printing Objects

Printing objects give us information about objects we are working with. In C++, we can do this by adding a friend ostream& operator << (ostream&, const Foobar&) method for the class. In Java, we use toString() method.
In python, this can be achieved by using __repr__ or __str__ methods.

Python

```
class Test:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return "Test a:%s b:%s" % (self.a, self.b)

    def __str__(self):
        return "From str method of Test: a is %s," \
               "b is %s" % (self.a, self.b)

# Driver Code
t = Test(1234, 5678)
print(t) # This calls __str__()
print([t]) # This calls __repr__()
```

Output :

```
From str method of Test: a is 1234,b is 5678
[Test a:1234 b:5678]
```

Important Points about Printing:

- If no __str__ method is defined, print t (or print str(t)) uses __repr__.

Python

```
class Test:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return "Test a:%s b:%s" % (self.a, self.b)

# Driver Code
t = Test(1234, 5678)
```

```
print(t)
```

Output :

```
Test a:1234 b:5678
```

- If no __repr__ method is defined then the default is used.

Python

```
class Test:
    def __init__(self, a, b):
        self.a = a
        self.b = b

# Driver Code
t = Test(1234, 5678)
print(t)
```

Output :

```
<__main__.Test instance at 0x7fa079da6710>
```

Whether you're preparing for your first job interview or aiming to upskill in this ever-evolving tech landscape, GeeksforGeeks Courses are your key to success. We provide top-quality content at affordable prices, all geared towards accelerating

your growth in a time-bound manner. Join the millions we've already empowered, and we're here to do the same for you. Don't miss out - check it out now!

Last Updated : 07 Jun, 2022

129

Previous

CBSE Class 11 | Computer Science - Python Syllabus

Next

Iterators in Python

Similar Reads

Data Hiding in Python

8 Tips For Object-Oriented Programming in Python

Introduction of Object Oriented Programming

Hiding and encrypting passwords in Python?

Python PyQt5 - Hiding the Progress Bar ?

PyQt5 - Hiding the radio button

PyQt5 QSpinBox – Hiding the spin box

PyQt5 QSpinBox – Hiding it using setHidden method

PyQt5 QCalendarWidget - Hiding according to the user

PyQt5 QCalendarWidget - Hiding the Navigation Bar

Complete Tutorials

OpenAI Python API - Complete Guide

Coding For Kids - Online Free Tutorial to Learn Coding

Computer Science and Programming For Kids

Pandas AI: The Generative AI Python Library

Python for Kids - Fun Tutorial to Learn Python Programming

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Improved By : sg4ipiafwot258z3lh6xa2mjq2qtxd89f49zgt7g

Article Tags : Python , School Programming

Practice Tags : python

Improve Article | Report Issue

GeeksforGeeks
Sanchhaya Education Private Limited
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

NCERT Solutions for Class 10
NCERT Solutions for Class 9
NCERT Solutions for Class 8
Complete Study Material
Social Science
English Grammar
Microeconimics
Statistics for Economics
Economics
Science and Technology Notes
Economics Notes
Important Topics in Ethics
UPSC Previous Year Papers
IBPS PO Syllabus
IBPS Clerk Syllabus
SSC CGL Practice Papers

Colleges

Indian Colleges Admission & Campus Experiences
Top Engineering Colleges
Top BCA Colleges
Top MBA Colleges
Top Architecture College
Choose College For Graduation

Companies

IT Companies
Software Development Companies
Artificial Intelligence(AI) Companies
CyberSecurity Companies
Service Based Companies
Product Based Companies
PSUs for CS Engineers

Preparation Corner

Company Wise Preparation
Preparation for SDE
Experienced Interviews
Internship Interviews
Competitive Programming
Aptitude Preparation
Puzzles

Exams

JEE Mains
JEE Advanced
GATE CS
NEET
UGC NET
CAT

More Tutorials

Software Testing
Software Development
Product Management
SAP
SEO
Linux
Excel

Write & Earn

Write an Article
Improve an Article
Pick Topics to Write
Share your Experiences
Internships

Python OOPs Concepts

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

Creating an Empty Class in Python

In the above example, we have created a class named Dog using the class keyword.

Python

```
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

Python Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

Python3

```
obj = Dog()
```

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Note: For more information, refer to self in the Python class

The Python __init__ Method

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Now let us define a class and create some objects using the self and __init__ method.

Creating a class and object with class and instance attributes

Python3

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
```

```
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.
  The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The __init__ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

Python3

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
```

```
        def __init__(self, name):
            self.name = name

        def speak(self):
            print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

Output

```
My name is Rodger
My name is Tommy
```

Note: For more information, refer to Python Classes and Objects

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Inheritance in Python

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

Python3

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

Output

```
Rahul
886012
My name is Rahul
IdNumber: 886012
Post: Intern
```

Note: For more information, refer to our Inheritance in Python tutorial.

Python Polymorphism

Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

Polymorphism in Python

This code demonstrates the concept of inheritance and method overriding in Python classes. It shows how subclasses can override methods defined in their parent class to provide specific behavior while still inheriting other methods from the parent class.

Python3

```
class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
```

```
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output

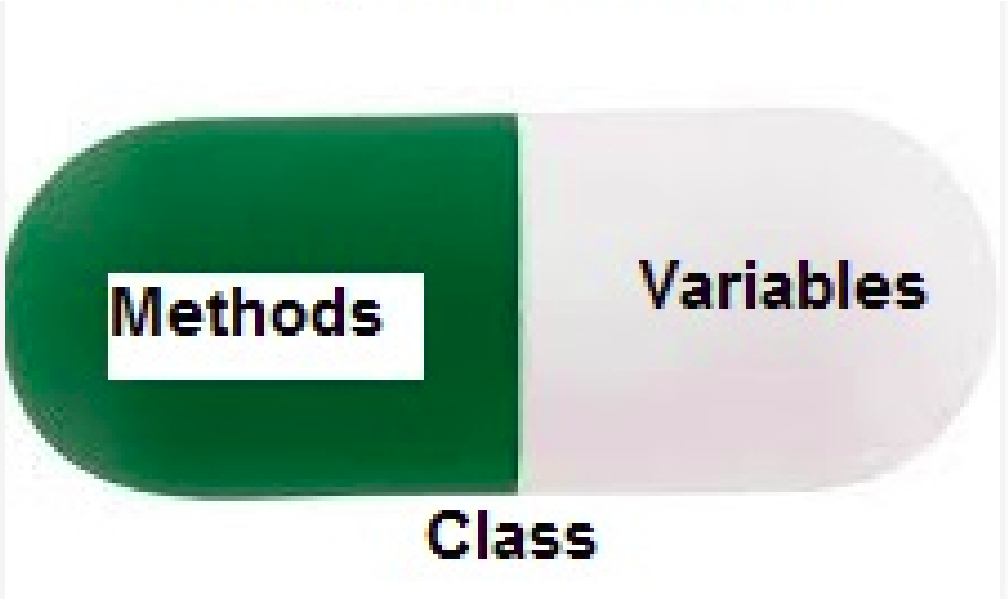
```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Note: For more information, refer to our Polymorphism in Python Tutorial.

Python Encapsulation

and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Encapsulation in Python

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can't even change its value.

Python3

```
# Python program to
# demonstrate private members

# Creating a Base class
class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a)

# Uncommenting print(obj1.c) will
# raise an AttributeError

# Uncommenting obj2 = Derived() will
# also raise an AtrributeError as
# private member of base class
# is called inside derived class
```

Output

```
GeeksforGeeks
```

Note: for more information, refer to our Encapsulation in Python Tutorial.

Data Abstraction

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.

Object Oriented Programming in Python | Set 2 (Data Hiding and Object Printing)

Whether you're preparing for your first job interview or aiming to upskill in this ever-evolving tech landscape, GeeksforGeeks Courses are your key to success. We provide top-quality content at affordable prices, all geared towards accelerating your growth in a time-bound manner. Join the millions we've already empowered, and we're here to do the same for you. Don't miss out - check it out now!

Last Updated : 23 Aug, 2023

539

Previous

Memoization using decorators in Python

Next

Python Classes and Objects

Similar Reads

Shuffle a deck of card with OOPS in Python

Python OOPS - Aggregation and Composition

30 OOPs Interview Questions and Answers (2023)

Top 10 Advance Python Concepts That You Must Know

Word Prediction using concepts of N - grams and CDF

Important differences between Python 2.x and Python 3.x with examples

Python program to build flashcard using class in Python

Python | Sort Python Dictionaries by Key or Value

Python | Merge Python key values to list

Reading Python File-Like Objects from C | Python

Complete Tutorials

OpenAI Python API - Complete Guide

Pandas AI: The Generative AI Python Library

Python for Kids - Fun Tutorial to Learn Python Programming

Data Analysis Tutorial

Flask Tutorial

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Improved By : Kushagra vatsa, nikhilaggarwal3, raj095907, tripathipriyanshu1998, surajkr_gupta, anushk6zi0

Article Tags : python-oop-concepts , Python

Practice Tags : python

Improve Article | Report Issue

Company

- About Us
- Legal
- Careers
- In Media
- Contact Us
- Advertise with us
- GFG Corporate Solution
- Placement Training Program
- Apply for Mentor

Explore

- Job-A-Thon Hiring Challenge
- Hack-A-Thon
- GfG Weekly Contest
- Offline Classes (Delhi/NCR)
- DSA in JAVA/C++
- Master System Design
- Master CP
- GeeksforGeeks Videos

Languages

- Python
- Java
- C++
- PHP
- GoLang
- SQL
- R Language
- Android Tutorial

DSA

- Data Structures
- Algorithms
- DSA for Beginners
- Basic DSA Problems
- DSA Roadmap
- Top 100 DSA Interview Problems
- DSA Roadmap by Sandeep Jain
- All Cheat Sheets

Data Science & ML

- Data Science With Python
- Data Science For Beginner
- Machine Learning Tutorial
- Maths For Machine Learning
- Pandas Tutorial
- NumPy Tutorial
- Deep Learning Tutorial

HTML & CSS

- HTML
- CSS
- Bootstrap
- Tailwind CSS
- SASS
- LESS
- Web Design

Computer Science

- GATE CS Notes
- Operating Systems
- Computer Network
- Database Management System
- Software Engineering
- Digital Logic Design
- Engineering Maths

Python

- Python Programming Examples
- Django Tutorial
- Python Projects
- Python Tkinter
- OpenCV Python Tutorial
- Python Interview Question

DevOps

- Git
- AWS
- Docker
- Kubernetes
- Azure
- GCP
- DevOps Roadmap

Competitive Programming

- Top DS or Algo for CP
- Top 50 Tree
- Top 50 Graph
- Top 50 Array
- Top 50 String
- Top 50 DP
- Top 15 Websites for CP

System Design

- What is System Design
- Monolithic and Distributed SD
- High Level Design or HLD
- Low Level Design or LLD
- Crack System Design Round
- System Design Interview Questions
- Grokking Modern System Design

JavaScript

- TypeScript
- ReactJS
- NextJS
- AngularJS
- NodeJS
- Express.js
- Lodash
- Web Browser

NCERT Solutions

- NCERT Solutions for Class 12
- NCERT Solution for Class 11
- NCERT Solutions for Class 10
- NCERT Solutions for Class 9
- NCERT Solutions for Class 8
- Complete Study Material

School Subjects

- Mathematics
- Physics
- Chemistry
- Biology
- Social Science
- English Grammar

Commerce

- Accountancy
- Business Studies
- Indian Economics
- Macroeconomics
- Microeconimics
- Statistics for Economics

Management & Finance

- Management
- HR Managament
- Income Tax
- Finance
- Economics

UPSC

- Polity Notes
- Geography Notes
- History Notes
- Science and Technology Notes
- Economics Notes
- Important Topics in Ethics
- UPSC Previous Year Papers

SSC/ BANKING

- SSC CGL Syllabus
- SBI PO Syllabus
- SBI Clerk Syllabus
- IBPS PO Syllabus
- IBPS Clerk Syllabus
- SSC CGL Practice Papers

Colleges

- Indian Colleges Admission & Campus Experiences
- Top Engineering Colleges
- Top BCA Colleges
- Top MBA Colleges
- Top Architecture College
- Choose College For Graduation

Companies

- IT Companies
- Software Development Companies
- Artificial Intelligence(AI) Companies
- CyberSecurity Companies
- Service Based Companies
- Product Based Companies
- PSUs for CS Engineers

Preparation Corner

- Company Wise Preparation
- Preparation for SDE
- Experienced Interviews
- Internship Interviews
- Competitive Programming
- Aptitude Preparation
- Puzzles

Exams

- JEE Mains
- JEE Advanced
- GATE CS
- NEET
- UGC NET
- CAT

More Tutorials

- Software Testing
- Software Development
- Product Management
- SAP
- SEO
- Linux
- Excel

Write & Earn

- Write an Article
- Improve an Article
- Pick Topics to Write
- Share your Experiences
- Internships

PYTHON OBJECT ORIENTED

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented *OO* programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming *OOP* to bring you at speed −

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members *classvariablesandinstancevariables* and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation:** The creation of an instance of a class.
- **Method :** A special kind of function that is defined in a class definition.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members *classvariablesandinstancevariables* and methods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class −

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together −

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
```

```
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result −

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time −

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions −

- The **getattr***obj, name*[*, default*] : to access the attribute of object.
- The **hasattr***obj, name* : to check if an attribute exists or not.
- The **setattr***obj, name, value* : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr***obj, name* : to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes −

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
```

```
      print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result −

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Destroying Objects *GarbageCollection*

Python deletes unneeded objects *built − intypesorclassinstances* automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container *list*, *tuple*, *ordictionary*. The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method __*del*__, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

Example

This __del__ destructor prints the class name of an instance that is about to be destroyed −

```
#!/usr/bin/python

class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
```

```
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

When the above code is executed, it produces following result −

```
3083401324 3083401324 3083401324
Point destroyed
```

Note: Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement.

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

Example

```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result −

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....

class B:         # define your calss B
.....

class C(A, B):   # subclass of A and B
.....
```

You can use issubclass or isinstance functions to check a relationships of two classes and instances.

- The **issubclass***sub, sup* boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance***obj, Class* boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

```
#!/usr/bin/python

class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result −

```
Calling child method
```

Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

SN	Method, Description & Sample Call
1	**__init__** *self[, args...]* Constructor *withanyoptionalarguments* Sample Call : *obj = classNameargs*
2	**__del__***self* Destructor, deletes an object Sample Call : *del obj*

3 **__repr__*self***
Evaluatable string representation
Sample Call : *repr*obj

4 **__str__*self***
Printable string representation
Sample Call : *str*obj

5 **__cmp__ *self, x***
Object comparison
Sample Call : *cmp*obj, x

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation −

Example

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

When the above code is executed, it produces the following result −

```
Vector(7,8)
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

Example

```
#!/usr/bin/python

class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

When the above code is executed, it produces the following result −

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you −

```
.......................
print counter._JustCounter__secretCount
```

When the above code is executed, it produces the following result −

```
1
2
2
```

Python Cheat Sheet

Quick Reference for Top Functions and Commands

Python Syntax Basics

Comments

Comments are an important part of your code, as they allow you to explain your thought process and make your code more readable. In Python, you can create single-line comments using the hash symbol (#).

```
# This is a single-line comment.
```

For multi-line comments, you can use triple quotes (either single or double).

```
""" This is a multi-line comment. """
```

Variables

Variables in Python are used to store data. You can assign values to variables using the equals sign (=).

```
x = 5
name = "John"
```

Variable names should be descriptive and follow the naming convention of using lowercase letters and underscores for spaces.

```
user_age = 25
favorite_color = "blue"
```

Data Types

The Python language comes with several data types built-in by default. Some of the more common ones include:

- **TEXT TYPES:** str
- **BOOLEAN TYPE:** bool
- **NUMERIC TYPES:** int, float, complex
- **SEQUENCE TYPES:** list, tuple, range
- **NONE TYPE:** Nonetype

To find out the data type of any Python object, you can use the type() function. For example:

```
name = 'jane'
print(type(name))
#Output: 'str'
```

Conditional Statements

Conditional statements in Python allow you to execute different codes based on certain conditions. The common conditional statements are **'if'**, **'elif'**, and **'else'**.

```
if condition:
    # Code to execute if the condition is true
elif another_condition:
    # Code to execute if the another_condition is true
else:
    # Code to execute if none of the conditions are true
```

Loops

A loop is used to repeatedly execute a block of code. Python has two types of loops: a **'for'** loop and a **'while'** loop.

Let's take a look at both of them:

For loops:

```
for variable in iterable:
    # Code to execute for each element in the iterable
```

While loops:

```
while condition:
    # Code to execute while the condition is true
```

Inside these loops, you can use conditional and control statements to control your program's flow.

Functions

Functions in Python are blocks of code that perform specific tasks. You can define a function using the **'def'** keyword, followed by the function name and parentheses containing any input parameters.

```
def function_name(parameters):
    # Code to execute
    return result
```

To call a function, use the function name followed by parentheses containing the necessary arguments.

```
function_name(arguments)
```

Now that we've gone over the Python basics, let's move on to some more advanced topics.

Data Structures

Lists

A list in Python is a mutable, ordered sequence of elements. To create a list, use square brackets and separate the elements with commas.

Python lists can hold a variety of data types like strings, integers, booleans, etc. Here are some examples of operations you can perform with Python lists:

- Create a list:

```
my_list = [1, 2, 3]
```

- Access elements:

```
my_list[0]
```

- Add an element:

```
my_list.append(4)
```

Tuples

A tuple is similar to a list, but it is immutable, which means you cannot change its elements once created. You can create a tuple by using parentheses and separating the elements with commas.

Here are some examples of tuple operations:

- Create a tuple:

```
my_tuple = (1, 2, 3)
```

- Access elements:

```
my_tuple[0] #Output: 1
```

Sets

A set is an unordered collection of unique elements. You can create a set using the set() function or curly braces.

It can also hold a variety of data types, as long as they are unique. Here are some examples of set operations:

- Create a set:

```
my_set = {1, 2, 3}
```

- Add an element:

```
my_set.add(4)
```

- Remove an element:

```
my_set.remove(1)
```

Dictionaries

A dictionary is an unordered collection of key-value pairs, where the keys are unique. You can create a dictionary using curly braces and separating the keys and values with colons. Here are some examples of dictionary operations:

- Create a dictionary:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
```

- Access elements:

```
my_dict['key1'] #Output:'value1'
```

- Add a key-value pair:

```
my_dict['key3'] = 'value3'
```

- Remove a key-value pair:

```
del my_dict['key1']
```

Remember to practice and explore these data structures in your Python projects to become more proficient in their usage.

File I/O

Reading Files

To read a file, you first need to open it using the built-in **open()** function, with the mode parameter set to **'r'** for reading:

```
file_obj = open('file_path', 'r')
```

Now that your file is open, you can use different methods to read its content:

- **read():** Reads the entire content of the file.
- **readline():** Reads a single line from the file.
- **readlines():** Returns a list of all lines in the file.

It's important to remember to close the file once you've finished working with it:

```
file_obj.close()
```

Alternatively, you can use the with statement, which automatically closes the file after the block of code completes:

```
with open('file_path', 'r') as file_obj:
    content = file_obj.read()
```

Writing Files

To create a new file or overwrite an existing one, open the file with mode **'w'**:

```
file_obj = open('file_path', 'w')
```

Write data to the file using the **write()** method:

```
file_obj.write('This is a line of text.')
```

Don't forget to close the file:

```
file_obj.close()
```

Again, consider using the with statement for a more concise and safer way to handle files:

```
with open('file_path', 'w') as file_obj:
    file_obj.write('This is a line of text.')
```

Appending To Files

To add content to an existing file without overwriting it, open the file with mode **'a'**:

```
file_obj = open('file_path', 'a')
```

Use the **write()** method to append data to the file:

```
file_obj.write('This is an extra line of text.')
```

And, as always, close the file when you're done:

```
file_obj.close()
```

For a more efficient and cleaner approach, use the with statement:

```
with open('file_path', 'a') as file_obj:
    file_obj.write('This is an extra line of text.')
```

By following these steps and examples, you can efficiently navigate file operations in your Python applications. Remember to always close your files after working with them to avoid potential issues and resource leaks.

Error Handling

Try And Except

To handle exceptions in your code, you can use the try and except blocks. The **try** block contains the code that might raise an error, whereas the except block helps you handle that exception, ensuring your program continues running smoothly.

Here's an example:

```
try:
    quotient = 5 / 0
except ZeroDivisionError as e:
    print("Oops! You're trying to divide by zero.")
```

In this case, the code inside the try block will raise a ZeroDivisionError exception. Since we have an except block to handle this specific exception, it will catch the error and print the message to alert you about the issue.

Finally

The **finally** block is used when you want to ensure that a specific block of code is executed, no matter the outcome of the try and except blocks. This is especially useful for releasing resources or closing files or connections, even if an exception occurs, ensuring a clean exit.

Here's an example:

```
try:
    # Your code here
except ZeroDivisionError as e:
    # Exception handling
finally:
    print("This will run no matter the outcome of the
    try and except blocks.")
```

Raising Exceptions

You can also raise custom exceptions in your code to trigger error handling when specific conditions are met. To do this, you can use the **raise** statement followed by the exception you want to raise (either built-in or custom exception).

For instance:

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be a negative value.")
try:
    validate_age(-3)
except ValueError as ve:
    print(ve)
```

In this example, we've defined a custom function to validate an age value. If the provided age is less than zero, we raise a **ValueError** with a custom message. When calling this function, you should wrap it in a try-except block to handle the exception properly.

Modules And Packages

Importing Modules

Modules in Python are files containing reusable code, such as functions, classes, or variables. Python offers several modules and packages for different tasks like data science, machine learning, robotics, etc.

To use a module's contents in your code, you need to import it first. Here are a few different ways to import a module:

- **import <module_name>:** This imports the entire module, and you can access its contents using the syntax **'module_name.content_name.'**

  For example:

  ```
  import random
  c = random.ranint()
  ```

- **from <module_name> import <content_name>:**

  This imports a specific content (function or variable) from the module, and you can use it directly without referencing the module name.

  ```
  from math import sin
  c = sin(1.57)
  ```

- **from <module_name> import *:** This imports all contents of the module. Be careful with this method as it can lead to conflicts if different modules have contents with the same name.

Some commonly used built-in Python modules include:

1. **math:** Provides mathematical functions and constants
2. **random:** Generates random numbers and provides related functions
3. **datetime:** Handles date and time operations
4. **os:** Interacts with the operating system and manages files and directories

Creating Packages

Packages in Python are collections of related modules. They help you organize your code into logical and functional units. To create a package:

1. Create a new directory with the desired package name.
2. Add an empty file named **init.py** to the directory. This file indicates to Python that the directory should be treated as a package.
3. Add your module files (with the .py extension) to the directory.

Now, you can import the package or its modules into your Python scripts. To import a module from a package, use the syntax:

```
import <package_name.module_name>
```

Structure your code with modules and packages to make it more organized and maintainable. This will also make it easier for you and others to navigate and comprehend your codebase.

Object-Oriented Programming

Classes

A class is a blueprint for creating objects. It defines the data (attributes) and functionality (methods) of the objects. To begin creating your own class, use the **"class"** keyword followed by the class name:

```
class ClassName:
    # Class attributes and methods
```

To add attributes and methods, simply define them within the class block. For example:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
    def bark(self):
        print("Woof!")
```

In this example, a new Dog object can be created with a name and breed, and it has a bark method that prints **"Woof!"** when called.

Inheritance

Inheritance allows one class to inherit attributes and methods from another class, enabling code reusability and modularity. The class that inherits is called a subclass or derived class, while the class being inherited from is called the base class or superclass.

To implement inheritance, add the name of the superclass in parentheses after the subclass name:

```
class SubclassName(SuperclassName):
    # Subclass attributes and methods
```

For instance, you could create a subclass "Poodle" from a "Dog" class:

```
class Poodle(Dog):
    def show_trick(self):
        print("The poodle does a trick.")
```

A Poodle object would now have all the attributes and methods of the Dog class, as well as its own show_trick method.

Encapsulation

Encapsulation is the practice of wrapping data and methods that operate on that data within a single unit, an object in this case. This promotes a clear separation between an object's internal implementation and its external interface.

Python employs name mangling to achieve encapsulation for class members by adding a double underscore prefix to the attribute name, making it seemingly private.

```
class Example:
    def __init__(self):
        self.__private_attribute = "I'm private!"
    def __private_method(self):
        print("You can't see me!")
```

Although you can still technically access these private members in Python, doing so is strongly discouraged as it violates encapsulation principles.

By understanding and implementing classes, inheritance, and encapsulation in your Python programs, you can utilize the power and flexibility of Object-Oriented Programming to create clean, modular, and reusable code.

Helpful Python Libraries

NumPy

NumPy is a popular Python library for mathematical and scientific computing. With its powerful N-dimensional array object, you can handle a wide range of mathematical operations, such as:

- Basic mathematical functions
- Linear algebra
- Fourier analysis
- Random number generation

NumPy's efficient array manipulations make it particularly suitable for projects that require numerical calculations.

Pandas

Pandas is a powerful data analysis and manipulation library that you can use to work with structured data. It's also very popular in the data science community due to the wide array of tools it provides for handling data.

Some of its features include:

- Data structures like Series (1D) and DataFrame (2D)
- Data cleaning and preparation
- Statistical analysis
- Time series functionality

By utilizing Pandas, you can easily import, analyze, and manipulate data in a variety of formats, such as CSV, Excel, and SQL databases. If you're interested in Pandas, you can check out our video on **How To Resample Time Series Data Using Pandas To Enhance Analysis:**

Youtube Reference

Requests

The Requests library simplifies the process of handling HTTP requests in Python. With this library, you can easily send and receive HTTP requests, such as GET, POST, and DELETE. Some key features include:

- Handling redirects and following links on web pages
- Adding headers, form data, and query parameters via simple Python libraries
- Managing cookies and sessions

Using Requests, you can quickly and efficiently interact with various web services and APIs.

Beautiful Soup

Beautiful Soup is a Python library for web scraping, which allows you to extract data from HTML and XML documents. Some of its key features include:

- Searching for specific tags or CSS classes
- Navigating and modifying parsed trees
- Extracting relevant information based on tag attributes

By using Beautiful Soup in conjunction with Requests, you can create powerful web scraping applications that gather information from a wide array of websites.

Python Datetime Module

The Datetime module allows us to work with date and time objects. It provides three additional data types: `date`, `time` and `datetime`.

```
import datetime
```

date()

```
datetime.date(year: int, month: int, day: int)
```

The date method return a date object with the `year`, `month` and `day` attributes:

```
>>> from datetime import date
>>> obj = date(2022, 12, 1)
>>> obj.year
# 2022
>>> obj.month
# 12
>>> obj.day
# 1
```

time()

```
datetime.time(hour: int, minute: int, second: int)
```

The `time` method return a time object with the `hour`, `minute`, `second`, `microsecond` and `tzinfo` attributes:

```
>>> from datetime import time
>>> obj = time(10, 20, 33)
>>> obj.hour
# 10
>>> obj.second
# 33
>>> obj.microsecond
# 0
```

datetime()

```
datetime.datetime(year, month, day, hour, minute, second)
```

The `datetime` returns an object with both, the `date` and `time` objects attributes:

```
>>> from datetime import datetime
>>> obj = datetime(2024, 12, 1, 15, 35, 59)
>>> obj.year
# 2024
>>> obj.month
# 12
>>> obj.day
# 1
>>> obj.hour
# 15
>>> obj.second
# 59
```

now() and today()

`now` and `today` methods return a `datetime` object with system's exact day and time:

```
>>> from datetime import datetime
>>> now = datetime.now()
```

```
>>> now
# datetime.datetime(2022, 7, 23, 19, 56, 49, 589806)
```

Because the object returned is a `datetime`, we can access both, `date` and `time` attributes:

```
>>> now.date()
# datetime.date(2022, 7, 23)
>>> now.time()
# datetime.time(19, 56, 49, 589806)
>>> now.year
# 2022
>>> now.month
# 7
>>> now.day
# 23
>>> now.hour
# 19
>>> now.minute
# 56
>>> now.second
# 49
>>> now.microsecond
# 589806
```

Additionally, `now` can take a **timezone** object as an optional parameter:

```
>>> from datetime import datetime, timezone
>>> datetime.now(timezone.utc)
# datetime.datetime(2022, 7, 24, 0, 20, 8, 265634, tzinfo=datetime.ti
```

If a *timezone* parameter is not specified, `now` will default to the system timezone.

strftime() and strptime()

You can easily transform between strings and datetime objects with the `strftime` and `strptime` methods.

strftime()

`strftime` allow us to create human formatted strings out of a Python datetime object:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
# datetime.datetime(2022, 7, 23, 20, 31, 19, 751479)

>>> now.strftime("%d-%b-%Y")
# '23-Jul-2022'

>>> now.strftime("%d-%m-%Y")
# '23-07-2022'

>>> now.strftime("%d-%b-%Y")
# '23-Jul-2022'

>>> now.strftime("%d-%m-%Y")
# '23-07-2022'

>>> now.strftime("%m/%d/%Y")
# '07/23/2022'

>>> now.strftime("%b/%d/%Y - %H:%M:%S")
# 'Jul/23/2022 - 20:31:19'
```

You may find the strings passed to `strftime` to be a little strange, but it is pretty easy to understand its meaning. For example, `%m/%d/%Y` will return the month, day, and year separated by `/` (07/23/2022).

strptime()

The `strptime` method creates a `datetime` object from a string.

This method accepts two parameters:

```
obj.strptime(datetime_string, format)
```

- A string representing a datetime object.

- The python format code equivalent to that string.

```
>>> from datetime import datetime

>>> datetime_str = '12-Jul-2023'
>>> datetime.strptime(datetime_str, '%d-%b-%Y')
# datetime.datetime(2023, 7, 12, 0, 0)

>>> datetime_str = 'Jul/12/2023 - 14:38:37'
>>> datetime.strptime(datetime_str, "%b/%d/%Y - %H:%M:%S")
# datetime.datetime(2023, 7, 12, 14, 38, 37)
```

Format Codes

Directive	Meaning	Example
`%a`	Weekday as locale's abbreviated name.	Sun, Mon, …, Sat (en_US)
`%A`	Weekday as locale's full name.	Sunday, Monday, …, Saturday (en_US)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6
`%d`	Day of the month as a zero-padded decimal number.	01, 02, …, 31
`%b`	Month as locale's abbreviated name.	Jan, Feb, …, Dec (en_US)
`%B`	Month as locale's full name.	January, February, …, December (en_US)
`%m`	Month as a zero-padded decimal number.	01, 02, …, 12
`%y`	Year without century as a zero-padded decimal number.	00, 01, …, 99
`%Y`	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23

Directive	Meaning	Example
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12
`%p`	Locale's equivalent of either AM or PM.	AM, PM (en_US)
`%M`	Minute as a zero-padded decimal number.	00, 01, …, 59
`%S`	Second as a zero-padded decimal number.	00, 01, …, 59
`%f`	Microsecond as a decimal number, zero-padded on the left.	000000, 000001, …, 999999
`%z`	UTC offset in the form `±HHMM[SS[.ffffff]]` (empty string if the object is naive).	(empty), +0000, -0400, +1030, +063415, -030712.345216
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, GMT
`%j`	Day of the year as a zero-padded decimal number.	001, 002, …, 366
`%U`	Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, …, 53
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, …, 53
`%c`	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US)
`%x`	Locale's appropriate date representation.	08/16/88 (None)
`%X`	Locale's appropriate time representation.	21:30:00 (en_US)
`%%`	A literal `'%'` character.	%

timedelta()

The `timedelta` object represents the difference between two dates or times.

```
>>> from datetime import datetime

>>> date_1 = datetime.strptime('12-Jul-2023', '%d-%b-%Y')
>>> date_2 = datetime.strptime('01-Jan-2024', '%d-%b-%Y')

>>> difference = date_2 - date_1
>>> difference
# datetime.timedelta(days=173)
>>> difference.days
# 173
```

`timedelta` can add `days`, `seconds` and `microseconds` to a datetime object:

```
>>> from datetime import datetime, timedelta

>>> now = datetime.now()
>>> now
# datetime.datetime(2022, 7, 23, 21, 25, 2, 341081)

>>> now + timedelta(days=10, seconds=15)
# datetime.datetime(2022, 8, 2, 21, 25, 17, 341081)
```

And can subtract `days`, `seconds` and `microseconds` to a datetime object:

```
>>> from datetime import datetime, timedelta

>>> now = datetime.now()
>>> now
# datetime.datetime(2022, 7, 23, 21, 25, 2, 341081)

>>> now - timedelta(days=10, seconds=15)
# datetime.datetime(2022, 7, 13, 21, 59, 41, 100883)
```

Previous page
Copy

Next page
Itertools

Subscribe to pythoncheatsheet.org

Join **8.800+ Python developers** in a two times a month and bullshit free **publication**, full of interesting, relevant links.

Your email address

SUBSCRIBE

Edit this page on GitHub

Do you have a question? ask the community

Do you see a bug? open an issue on GitHub

datacamp python

Working with Dates and Times in Python

Learn Python Basics online at www.DataCamp.com

> Key definitions

When working with dates and times, you will encounter technical terms and jargon such as the following:

- **Date:** Handles dates without time.
- **POSIXct:** Handles date & time in calendar time.
- **POSIXlt:** Handles date & time in local time.
- **Hms:** Parses periods with hour, minute, and second
- **Timestamp:** Represents a single pandas date & time
- **Interval:** Defines an open or closed range between dates and times
- **Time delta:** Computes time difference between different datetimes

> The ISO8601 datetime format

The **ISO 8601 datetime format** specifies datetimes from the largest to the smallest unit of time **(YYYY-MM-DD HH:MM:SS TZ).** Some of the advantages of ISO 8601 are:

- It avoids ambiguities between MM/DD/YYYY and DD/MM/YYYY formats.
- The 4-digit year representation mitigates overflow problems after the year 2099.
- Using numeric month values (08 not AUG) makes it language independent, so dates make sense throughout the world.
- Python is optimized for this format since it makes comparison and sorting easier.

> Packages used in this cheat sheet

Load the packages and dataset used in this cheatsheet.

```
import datetime as dt
import time as tm
import pytz
import pandas as pd
```

In this cheat sheet, we will be using 3 pandas series — `iso, us, non_us,` and 1 pandas DataFrame `parts`

iso
1969-07-20 20:17:40
1969-11-19 06:54:35
1971-02-05 09:18:11

us
07/20/1969 20:17:40
11/19/1969 06:54:35
02/05/1971 09:18:11

non_us
20/07/1969 20:17:40
19/11/1969 06:54:35
05/02/1971 09:18:11

parts		
year	month	day
1969	7	20
1969	11	19
1971	2	5

> Getting the current date and time

```
# Get the current date
dt.date.today()

# Get the current date and time
dt.datetime.now()
```

> Reading date, datetime, and time columns in a CSV file

```
# Specify datetime column
pd.read_csv("filename.csv", parse_dates = ["col1", "col2"])

# Specify datetime column
pd.read_csv("filename.csv", parse_dates = {"col1": ["year", "month", "day"]})
```

> Parsing dates, datetimes, and times

```
# Parse dates in ISO format
pd.to_datetime(iso)

# Parse dates in US format
pd.to_datetime(us, dayfirst=False)

# Parse dates in NON US format
pd.to_datetime(non_us, dayfirst=True)

# Parse dates, guessing a single format
pd.to_datetime(iso, infer_datetime_format=True)

# Parse dates in single, specified format
pd.to_datetime(iso, format="%Y-%m-%d %H:%M:%S")

# Parse dates in single, specified format
pd.to_datetime(us, format="%m/%d/%Y %H:%M:%S")

# Make dates from components
pd.to_datetime(parts)
```

> Extracting components

```
# Parse strings to datetimes
dttm = pd.to_datetime(iso)

# Get year from datetime pandas series
dttm.dt.year

# Get day of the year from datetime pandas series
dttm.dt.day_of_year

# Get month name from datetime pandas series
dttm.dt.month_name()

# Get day name from datetime pandas series
dttm.dt.day_name()

# Get datetime.datetime format from datetime pandas series
dttm.dt.to_pydatetime()
```

> Rounding dates

```
# Rounding dates to nearest time unit
dttm.dt.round('1min')

# Flooring dates to nearest time unit
dttm.dt.floor('1min')

# Ceiling dates to nearest time unit
dttm.dt.ceil('1min')
```

> Arithmetic

```
# Create two datetimes
now = dt.datetime.now()
then = pd.Timestamp('2021-09-15 10:03:30')

# Get time elapsed as timedelta object
now - then

# Get time elapsed in seconds
(now - then).total_seconds()

# Adding a day to a datetime
dt.datetime(2022,8,5,11,13,50) + dt.timedelta(days=1)
```

> Time Zones

```
# Get current time zone
tm.localtime().tm_zone

# Get a list of all time zones
pytz.all_timezones

# Parse strings to datetimes
dttm = pd.to_datetime(iso)

# Get datetime with timezone using location
dttm.dt.tz_localize('CET', ambiguous='infer')

# Get datetime with timezone using UTC offset
dttm.dt.tz_localize('+0100')

# Convert datetime from one timezone to another
dttm.dt.tz_localize('+0100').tz_convert('US/Central')
```

> Time Intervals

```
# Create interval datetimes
start_1 = pd.Timestamp('2021-10-21 03:02:10')
finish_1 = pd.Timestamp('2022-09-15 10:03:30')
start_2 = pd.Timestamp('2022-08-21 03:02:10')
finish_2 = pd.Timestamp('2022-12-15 10:03:30')

# Specify the interval between two datetimes
pd.Interval(start_1, finish_1, closed='right')

# Get the length of an interval
pd.Interval(start_1, finish_1, closed='right').length

# Determine if two intervals are intersecting
pd.Interval(start_1, finish_1,
closed='right').overlaps(pd.Interval(start_2, finish_2, closed='right'))
```

> Time Deltas

```
# Define a timedelta in days
pd.Timedelta(7, "d")

# Convert timedelta to seconds
pd.Timedelta(7, "d").total_seconds()
```

Data Science Cheat Sheet

Python Regular Expressions

SPECIAL CHARACTERS

`^` | Matches the expression to its right at the start of a string. It matches every such instance before each `\n` in the string.

`$` | Matches the expression to its left at the end of a string. It matches every such instance before each `\n` in the string.

`.` | Matches any character except line terminators like `\n`.

`\` | Escapes special characters or denotes character classes.

`A|B` | Matches expression `A` or `B`. If A is matched first, B is left untried.

`+` | Greedily matches the expression to its left 1 or more times.

`*` | Greedily matches the expression to its left 0 or more times.

`?` | Greedily matches the expression to its left 0 or 1 times. But if `?` is added to qualifiers (`+`, `*`, and `?` itself) it will perform matches in a non-greedy manner.

`{m}` | Matches the expression to its left `m` times, and not less.

`{m,n}` | Matches the expression to its left `m` to `n` times, and not less.

`{m,n}?` | Matches the expression to its left `m` times, and ignores `n`. See `?` above.

CHARACTER CLASSES [A.K.A. SPECIAL SEQUENCES]

`\w` | Matches alphanumeric characters, which means `a-z`, `A-Z`, and `0-9`. It also matches the underscore, `_`.

`\d` | Matches digits, which means `0-9`.

`\D` | Matches any non-digits.

`\s` | Matches whitespace characters, which include the `\t`, `\n`, `\r`, and space characters.

`\S` | Matches non-whitespace characters.

`\b` | Matches the boundary (or empty string) at the start and end of a word, that is, between `\w` and `\W`.

`\B` | Matches where `\b` does not, that is, the boundary of `\w` characters.

`\A` | Matches the expression to its right at the absolute start of a string whether in single or multi-line mode.

`\Z` | Matches the expression to its left at the absolute end of a string whether in single or multi-line mode.

SETS

`[ ]` | Contains a set of characters to match.

`[amk]` | Matches either `a`, `m`, or `k`. It does not match `amk`.

`[a-z]` | Matches any alphabet from `a` to `z`.

`[a\-z]` | Matches `a`, `-`, or `z`. It matches `-` because `\` escapes it.

`[a-]` | Matches `a` or `-`, because `-` is not being used to indicate a series of characters.

`[-a]` | As above, matches `a` or `-`.

`[a-z0-9]` | Matches characters from `a` to `z` and also from `0` to `9`.

`[(+*)]` | Special characters become literal inside a set, so this matches `(`, `+`, `*`, and `)`.

`[^ab5]` | Adding `^` excludes any character in the set. Here, it matches characters that are not `a`, `b`, or `5`.

GROUPS

`( )` | Matches the expression inside the parentheses and groups it.

`(?)` | Inside parentheses like this, `?` acts as an extension notation. Its meaning depends on the character immediately to its right.

`(?PAB)` | Matches the expression `AB`, and it can be accessed with the group name.

`(?aiLmsux)` | Here, `a`, `i`, `L`, `m`, `s`, `u`, and `x` are flags:

- `a` — Matches ASCII only
- `i` — Ignore case
- `L` — Locale dependent
- `m` — Multi-line
- `s` — Matches all
- `u` — Matches unicode
- `x` — Verbose

`(?:A)` | Matches the expression as represented by `A`, but unlike `(?PAB)`, it cannot be retrieved afterwards.

`(?#...)` | A comment. Contents are for us to read, not for matching.

`A(?=B)` | Lookahead assertion. This matches the expression `A` only if it is followed by `B`.

`A(?!B)` | Negative lookahead assertion. This matches the expression `A` only if it is not followed by `B`.

`(?<=B)A` | Positive lookbehind assertion. This matches the expression `A` only if `B` is immediately to its left. This can only matched fixed length expressions.

`(?<!B)A` | Negative lookbehind assertion. This matches the expression `A` only if `B` is not immediately to its left. This can only matched fixed length expressions.

`(?P=name)` | Matches the expression matched by an earlier group named "name".

`(...)\1` | The number `1` corresponds to the first group to be matched. If we want to match more instances of the same expression, simply use its number instead of writing out the whole expression again. We can use from `1` up to `99` such groups and their corresponding numbers.

POPULAR PYTHON RE MODULE FUNCTIONS

`re.findall(A, B)` | Matches all instances of an expression `A` in a string `B` and returns them in a list.

`re.search(A, B)` | Matches the first instance of an expression `A` in a string `B`, and returns it as a re match object.

`re.split(A, B)` | Split a string `B` into a list using the delimiter `A`.

`re.sub(A, B, C)` | Replace `A` with `B` in the string `C`.

Quick-Start: Regex Cheat Sheet

The tables below are a reference to basic regex. While reading the rest of the site, when in doubt, you can always come back and look here. (It you want a bookmark, here's a direct link to the regex reference tables). I encourage you to print the tables so you have a cheat sheet on your desk for quick reference.

The tables are not exhaustive, for two reasons. First, every regex flavor is different, and I didn't want to crowd the page with overly exotic syntax. For a full reference to the particular regex flavors you'll be using, it's always best to go straight to the source. In fact, for some regex engines (such as Perl, PCRE, Java and .NET) you may want to check once a year, as their creators often introduce new features.

The other reason the tables are not exhaustive is that I wanted them to serve as a quick introduction to regex. If you are a complete beginner, you should get a firm grasp of basic regex syntax just by reading the examples in the tables. I tried to introduce features in a logical order and to keep out oddities that I've never seen in actual use, such as the "bell character". With these tables as a jumping board, you will be able to advance to mastery by exploring the other pages on the site.

How to use the tables

The tables are meant to serve as an accelerated regex course, and they are meant to be read slowly, one line at a time. On each line, in the leftmost column, you will find a new element of regex syntax. The next column, "Legend", explains what the element means (or encodes) in the regex syntax. The next two columns work hand in hand: the "Example" column gives a valid regular expression that uses the element, and the "Sample Match" column presents a text string that could be matched by the regular expression.

You can read the tables online, of course, but if you suffer from even the mildest case of online-ADD (attention deficit disorder), like most of us… Well then, I highly recommend you print them out. You'll be able to study them slowly, and to use them as a cheat sheet later, when you are reading the rest of the site or experimenting with your own regular expressions.

Enjoy!

If you overdose, make sure not to miss the next page, which comes back down to Earth and talks about some really cool stuff: **The 1001 ways to use Regex**.

Regex Accelerated Course and Cheat Sheet

For easy navigation, here are some jumping points to various sections of the page:

(direct link)

Characters

Character	Legend	Example	Sample Match
\d	Most engines: one digit from 0 to 9	file_\d\d	file_25
\d	.NET, Python 3: one Unicode digit in any script	file_\d\d	file_9৩
\w	Most engines: "word character": ASCII letter, digit or underscore	\w-\w\w\w	A-b_1
\w	.Python 3: "word character": Unicode letter, ideogram, digit, or underscore	\w-\w\w\w	字-ま_۳
\w	.NET: "word character": Unicode letter, ideogram, digit, or connector	\w-\w\w\w	字-ま‿۳
\s	Most engines: "whitespace character": space, tab, newline, carriage return, vertical tab	a\sb\sc	a b c
\s	.NET, Python 3, JavaScript: "whitespace character": any Unicode separator	a\sb\sc	a b c
\D	One character that is not a *digit* as defined by your engine's *\d*	\D\D\D	ABC
\W	One character that is not a *word character* as defined by your engine's *\w*	\W\W\W\W\W	*-+=)
\S	One character that is not a *whitespace character* as defined by your engine's *\s*	\S\S\S\S	Yoyo

(direct link)

Quantifiers

Quantifier	Legend	Example	Sample Match
+	One or more	Version \w-\w+	Version A-b1_1
{3}	Exactly three times	\D{3}	ABC
{2,4}	Two to four times	\d{2,4}	156
{3,}	Three or more times	\w{3,}	regex_tutorial
*	Zero or more times	A*B*C*	AAACC
?	Once or none	plurals?	plural

(direct link)

More Characters

Character	Legend	Example	Sample Match
.	Any character except line break	a.c	abc
.	Any character except line break	.*	whatever, man.
\.	A period (special character: needs to be escaped by a \)	a\.c	a.c
\	Escapes a special character	\.*\+\? \$\^\/\\	.*+? $^/\
\	Escapes a special character	\[\{\(\)\}\]	[{()}]

(direct link)

Logic

Logic	Legend	Example	Sample Match
|	Alternation / OR operand	22|33	33
(…)	Capturing group	A(nt|pple)	Apple (captures "pple")
\1	Contents of Group 1	r(\w)g\1x	regex
\2	Contents of Group 2	(\d\d)\+(\d\d)=\2\+\1	12+65=65+12
(?: …)	Non-capturing group	A(?:nt|pple)	Apple

(direct link)

More White-Space

Character	Legend	Example	Sample Match
\t	Tab	T\t\w{2}	T ab
\r	Carriage return character	see below	
\n	Line feed character	see below	
\r\n	Line separator on Windows	AB\r\nCD	AB CD
\N	Perl, PCRE (C, PHP, R…): one character that is not a line break	\N+	ABC
\h	Perl, PCRE (C, PHP, R…), Java: one horizontal whitespace character: tab or Unicode space separator		
\H	One character that is not a horizontal whitespace		
\v	.NET, JavaScript, Python, Ruby: vertical tab		
\v	Perl, PCRE (C, PHP, R…), Java: one vertical whitespace character: line feed, carriage return, vertical tab, form feed, paragraph or line separator		
\V	Perl, PCRE (C, PHP, R…), Java: any character that is not a vertical whitespace		
\R	Perl, PCRE (C, PHP, R…), Java: one line break (carriage return + line feed pair, and all the characters matched by \v)		

(direct link)

More Quantifiers

Quantifier	Legend	Example	Sample Match
+	The + (one or more) is "greedy"	\d+	12345
?	Makes quantifiers "lazy"	\d+?	1 in **1**2345
*	The * (zero or more) is "greedy"	A*	AAA
?	Makes quantifiers "lazy"	A*?	empty in AAA
{2,4}	Two to four times, "greedy"	\w{2,4}	abcd
?	Makes quantifiers "lazy"	\w{2,4}?	ab in **ab**cd

(direct link)

Character Classes

Character	Legend	Example	Sample Match
[…]	One of the characters in the brackets	[AEIOU]	One uppercase vowel
[…]	One of the characters in the brackets	T[ao]p	*Tap* or *Top*
-	Range indicator	[a-z]	One lowercase letter
[x-y]	One of the characters in the range from x to y	[A-Z]+	GREAT
[…]	One of the characters in the brackets	[AB1-5w-z]	One of either: A,B,1,2,3,4,5,w,x,y,z
[x-y]	One of the characters in the range from x to y	[-~]+	Characters in the printable section of the ASCII table.
[^x]	One character that is not x	[^a-z]{3}	A1!
[^x-y]	One of the characters **not** in the range from x to y	[^ -~]+	Characters that are **not** in the printable section of the ASCII table.
[\d\D]	One character that is a digit or a non-digit	[\d\D]+	Any characters, including new lines, which the regular dot doesn't match
[\x41]	Matches the character at hexadecimal position 41 in the ASCII table, i.e. A	[\x41-\x45]{3}	ABE

(direct link)

Anchors and Boundaries

Anchor	Legend	Example	Sample Match
^	Start of string or start of line depending on multiline mode. (But when [^inside brackets], it means "not")	^abc .*	abc (line start)
$	End of string or end of line depending on multiline mode. Many engine-dependent subtleties.	.*? the end$	this is the end
\A	Beginning of string (all major engines except JS)	\Aabc[\d\D]*	abc (string... ...start)
\z	Very end of the string Not available in Python and JS	the end\z	this is...\n...**the end**
\Z	End of string or (except Python) before final line break Not available in JS	the end\Z	this is...\n...**the end**\n
\G	Beginning of String or End of Previous Match .NET, Java, PCRE (C, PHP, R…), Perl, Ruby		
\b	Word boundary Most engines: position where one side only is an ASCII letter, digit or underscore	Bob.*\bcat\b	Bob ate the cat
\b	Word boundary .NET, Java, Python 3, Ruby: position where one side only is a Unicode letter, digit or underscore	Bob.*\b\кошка\b	Bob ate the кошка
\B	Not a word boundary	c.*\Bcat\B.*	copycats

(direct link)

POSIX Classes

Character	Legend	Example	Sample Match
[:alpha:]	PCRE (C, PHP, R…): ASCII letters A-Z and a-z	[8[:alpha:]]+	WellDone88
[:alpha:]	Ruby 2: Unicode letter or ideogram	[[:alpha:]\d]+	кошка99
[:alnum:]	PCRE (C, PHP, R…): ASCII digits and letters A-Z and a-z	[[:alnum:]]{10}	ABCDE12345
[:alnum:]	Ruby 2: Unicode digit, letter or ideogram	[[:alnum:]]{10}	кошка90210
[:punct:]	PCRE (C, PHP, R…): ASCII punctuation mark	[[:punct:]]+	?!.,:;
[:punct:]	Ruby: Unicode punctuation mark	[[:punct:]]+	¿.: ~\¡

(direct link)

Inline Modifiers

None of these are supported in JavaScript. In Ruby, beware of (?s) and (?m).

Modifier	Legend	Example	Sample Match
(?i)	Case-insensitive mode (except JavaScript)	(?i)Monday	monDAY
(?s)	DOTALL mode (except JS and Ruby). The dot (.) matches new line characters (\r\n). Also known as "single-line mode" because the dot treats the entire input as a single line	(?s)From A.*to Z	From A to Z
(?m)	Multiline mode (except Ruby and JS) ^ and $ match at the beginning and end of every line	(?m)1\r\n^2$\r\n^3$	1 2 3
(?m)	In Ruby: the same as (?s) in other engines, i.e. DOTALL mode, i.e. dot matches line breaks	(?m)From A.*to Z	From A to Z
(?x)	Free-Spacing Mode mode (except JavaScript). Also known as comment mode or whitespace mode	(?x) # this is a # comment abc # write on multiple # lines [ ]d # spaces must be # in brackets	abc d

(?n)	.NET, PCRE 10.30+: named capture only	Turns all (parentheses) into non-capture groups. To capture, use named groups.
(?d)	Java: Unix linebreaks only	The dot and the ^ and $ anchors are only affected by \n
(?^)	PCRE 10.32+: unset modifiers	Unsets ismnx modifiers

(direct link)

Lookarounds

Lookaround	Legend	Example	Sample Match
(?=…)	Positive lookahead	(?=\d{10})\d{5}	01234 in **01234**56789
(?<=…)	Positive lookbehind	(?<=\d)cat	cat in 1**cat**
(?!…)	Negative lookahead	(?!theatre)the\w+	theme
(?<!…)	Negative lookbehind	\w{3}(?<!mon)ster	Munster

(direct link)

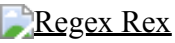
Character Class Operations

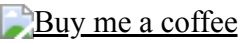
Class Operation	Legend	Example	Sample Match
[…-[…]]	.NET: character class subtraction. One character that is in those on the left, but not in the subtracted class.	[a-z-[aeiou]]	Any lowercase consonant
[…-[…]]	.NET: character class subtraction.	[\p{IsArabic}-[\D]]	An Arabic character that is not a non-digit, i.e., an Arabic digit
[…&&[…]]	Java, Ruby 2+: character class intersection. One character that is both in those on the left and in the && class.	[\S&&[\D]]	An non-whitespace character that is a non-digit.
[…&&[…]]	Java, Ruby 2+: character class intersection.	[\S&&[\D]&&[^a-zA-Z]]	An non-whitespace character that a non-digit and not a letter.
[…&&[^…]]	Java, Ruby 2+: character class subtraction is obtained by intersecting a class with a negated class	[a-z&&[^aeiou]]	An English lowercase letter that is not a vowel.
[…&&[^…]]	Java, Ruby 2+: character class subtraction	[\p{InArabic}&&[^\p{L}\p{N}]]	An Arabic character that is not a letter or a number

(direct link)

Other Syntax

Syntax	Legend	Example	Sample Match
\K	Keep Out Perl, PCRE (C, PHP, R…), Python's alternate *regex* engine, Ruby 2+: drop everything that was matched so far from the overall match to be returned	prefix\K\d+	12
\Q…\E	Perl, PCRE (C, PHP, R…), Java: treat anything between the delimiters as a literal string. Useful to escape metacharacters.	\Q(C++ ?)\E	(C++ ?)


Ask Rex



Leave a Comment
1-10 of 19 Threads
Appu – Japan
March 07, 2022 - 19:05
Subject: You are God of regex !! Thank you so much :)

This site is absolute gold mine. I once stumbled upon and missed it, now found again… So happy :D Thank you so much for all your efforts!!
maureen – san francisco
June 18, 2021 - 16:25
Subject: absolutely the BEST website for regex

This is the go-to website for everything on regex. Thank you!
Pythia – New Zealand
July 15, 2020 - 03:54
Subject: Very thoughtful and useful cheat sheet

Unlike lots of other cheat sheets or regex web sites, I was able (without much persistent regex knowledge) to apply the rules and to solve my problem. THANK YOU :)
Mark
July 04, 2020 - 10:14
Subject: Thanks a lot

Thanks a lot for the quick guide. It's really helpful.
Purusharth Amrut
June 10, 2020 - 14:41
Subject: Very useful site

Thank you soooooo much for this site. I'm using python regex for natural language processing in sentiment analysis and this helped me a lot.
Alessandro Maiorana – Italy, Milan
April 15, 2020 - 12:43
Subject: Thank you! Excellent resource for any student

Thank you so much for this incredible cheatsheet! It is facilitating a lot my regex learning! God bless you and your passion!
michael – Bulgaria
April 10, 2020 - 12:43
Subject: Thank you for doing such a geat work.

I am now learning regex and for finding such a well organized site is a blessing! You are a good soul! Thank you for everything and stay inspired!
Yuri – California
November 13, 2019 - 17:39
Subject: Simple = perfect

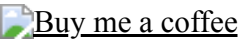
Thanks a lot, saved me tons of time!!!!
Tom – Europe, Poland
September 30, 2019 - 18:43
Subject: Congratulations

Well done, very useful page. Thank you for your effort. T
Najam
March 25, 2019 - 03:44
Subject: Thank you very much

Hi Rex,

Thankyou very much for compiling these. I am new to text analytics and is struggling a lot with regex. This is helping me a lot pick up. Great work


Next


Buy me a coffee

datacamp

Regular Expressions Cheat Sheet

Learn regular expressions online at www.DataCamp.com

What is a regular expression?

Regular expression (regex or regexp) is a pattern of characters that describes an amount of text. To process regexes, you will use a "regex engine." Each of these engines use slightly different syntax called regex flavor. A list of popular engines can be found here. Two common programming languages we discuss on DataCamp are Python and R which each have their own engines.

Since regex describes patterns of text, it can be used to check for the existence of patterns in a text, extract substrings from longer strings, and help make adjustments to text. Regex can be very simple to describe specific words, or it can be more advanced to find vague patterns of characters like the top-level domain in a url.

> Definitions

Literal character: A literal character is the most basic regular expression you can use. It simply matches the actual character you write. So if you are trying to represent an "r," you would write `r`.

Metacharacter: Metacharacters signify to the regex engine that the following character has a special meaning. You typically include a `\` in front of the metacharacter and they can do things like signify the beginning of a line, end of a line, or to match any single character.

Character class: A character class (or character set) tells the engine to look for one of a list of characters. It is signified by `[` and `]` with the characters you are looking for in the middle of the brackets.

Capture group: A capture group is signified by opening and closing, round parenthesis. They allow you to group regexes together to apply other regex features like quantifiers (see below) to the group.

> Anchors

Anchors match a position before or after other characters.

Syntax	Description	Example pattern	Example matches	Example non-matches
`^`	match start of line	`^r`	rabbit raccoon	parrot ferret
`$`	match end of line	`t$`	rabbit foot	trap star
`\A`	match start of line	`\Ar`	rabbit raccoon	parrot ferret
`\Z`	match end of line	`t\Z`	rabbit foot	trap star
`\b`	match characters at the start or end of a word	`\bfox\b`	the red fox ran the fox ate	foxtrot foxskin scarf
`\B`	match characters in the middle of other non-space characters	`\Bee\B`	trees beef	bee tree

> Matching types of character

Rather than matching specific characters, you can match specific types of characters such as letters, numbers, and more.

Syntax	Description	Example pattern	Example matches	Example non-matches
`.`	anything except for a linebreak	`c.e`	clean cheap	acert cent
`\d`	match a digit	`\d`	6060-842 2b|^2b	two **___
`\D`	match a non-digit	`\D`	The 5 cats ate 12 Angry men	52 10032

Syntax	Description	Example pattern	Example matches	Example non-matches
`\w`	match word characters	`\wee\w`	trees bee4	The bee eels eat meat
`\W`	match non-word characters	`\Wbat\W`	At bat Swing the bat fast	wombat bat53
`\s`	match whitespace	`\sfox\s`	the fox ate his fox ran	it's the fox. foxfur
`\S`	match non-whitespace	`\See\S`	trees beef	the bee stung The tall tree
`\metacharacter`	escape a metacharacter to match on the metacharacter	`\.` `\^`	The cat ate. 2^3	the cat ate 23

> Character classes

Character classes are sets or ranges of characters.

Syntax	Description	Example pattern	Example matches	Example non-matches
`[xy]`	match several characters	`gr[ea]y`	gray grey	green greek
`[x-y]`	match a range of characters	`[a-e]`	amber brand	fox join
`[^xy]`	does not match several characters	`gr[^ea]y`	green greek	gray grey
`[\^-]`	match metacharacters inside the character class	`4[\^\.-+*/]\d`	4^3 4.2	44 23

> Repetition

Rather than matching single instances of characters, you can match repeated characters.

Syntax	Description	Example pattern	Example matches	Example non-matches
`x*`	match zero or more times	`ar*o`	cacao carrot	arugula artichoke
`x+`	match one or more times	`re+`	green tree	trap ruined
`x?`	match zero or one times	`ro?a`	roast rant	root rear
`x{m}`	match m times	`\we{2}\w`	deer seer	red enter
`x{m,}`	match m or more times	`2{3,}4`	671-2224 2222224	224 123
`x{m,n}`	match between m and n times	`12{1,3}3`	1234 1222384	15335 1222223
`x*?, x+?, etc.`	match the minimum number of times - known as a lazy quantifier	`re+?`	tree freeeee	trout roasted

> Capturing, alternation & backreferences

In order to extract specific parts of a string, you can capture those parts, and even name the parts that you captured.

Syntax	Description	Example pattern	Example matches	Example non-matches
`(x)`	capturing a pattern	`(iss)+`	Mississippi missed	mist persist
`(?:x)`	create a group without capturing	`(?:ab)(cd)`	Match: abcd Group 1: cd	acbd
`(?<name>x)`	create a named capture group	`(?<first>\d)(?<scrond>\d)\d*`	Match: 1325 first: 1 second: 3	2 hello

Syntax	Description	Example pattern	Example matches	Example non-matches
`(x|y)`	match several alternative patterns	`(re|ba)`	red banter	rant bear
`\n`	reference previous captures where n is the group index starting at 1	`(b)(\w*)\1`	blob bribe	bear bring
`\k<name>`	reference named captures	`(?<first>5)(\d*)\k<first>`	51245 55	523 51

> Lookahead

You can specify that specific characters must appear before or after you match, without including those characters in the match.

Syntax	Description	Example pattern	Example matches	Example non-matches
`(?=x)`	looks ahead at the next characters without using them in the match	`an(?=an)` `iss(?=ipp)`	banana Mississippi	band missed
`(?!x)`	looks ahead at next characters to not match on	`ai(?!n)`	fail brail	faint train
`(?<=x)`	looks at previous characters for a match without using those in the match	`(?<=tr)a`	trail translate	bear streak
`(?<!x)`	looks at previous characters to not match on	`(?!tr)a`	bear translate	trail strained

> Literal matches and modifiers

Modifiers are settings that change the way the matching rules work.

Syntax	Description	Example pattern	Example matches	Example non-matches
`\Qx\E`	match start to finish	`\Qtell\E` `\Q\d\E`	tell \d	I'll tell you this I have 5 coins
`(?i)x(?-i).`	set the regex string to case-insensitive	`(?i)te(?-i)`	sTep tEach	Trench bear
`(?x)x(?-x)`	regex ignores whitespace	`(?x)t a p(?-x)`	tap tapdance	c a t rot a potato
`(?s)x(?-s)`	turns on single-line/ DOTALL mode which makes the "." include new-line symbols (\n) in addition to everything else	`(?s)first and second(?-s) and third`	first and Second and third	first and second and third
`(?m)x(?-m)`	changes ^ and $ to be end of line rather than end of string	`^eat and sleep$`	eat and sleep eat and sleep	treat and sleep eat and sleep.

> Unicode

Regular expressions can work beyond the Roman alphabet, with things like Chinese characters or emoji.

- **Code Points:** The hexadecimal number used to represent an abstract character in a system like unicode.
- **Graphemes:** Is either a codepoint or a character. All characters are made up of one or more graphemes in a sequence.

Syntax	Description	Example pattern	Example matches	Example non-matches
`\X`	match graphemes	`\u0000gmail`	@gmail www.email@gmail	gmail @aol
`\X\X`	match special characters like ones with an accent	`\u00e8 or \u0065\u0300`	è	e

DASK FOR PARALLEL COMPUTING CHEAT SHEET

See full Dask documentation at: **http://dask.pydata.org/**

These instructions use the conda environment manager. Get yours at **http://bit.ly/getconda**

DASK QUICK INSTALL	
Install Dask with conda	`conda install dask`
Install Dask with pip	`pip install dask[complete]`
DASK COLLECTIONS	**EASY TO USE BIG DATA COLLECTIONS**
DASK DATAFRAMES	PARALLEL PANDAS DATAFRAMES FOR LARGE DATA
Import	`import dask.dataframe as dd`
Read CSV data	`df = dd.read_csv('my-data.*.csv')`
Read Parquet data	`df = dd.read_parquet('my-data.parquet')`
Filter and manipulate data with Pandas syntax	`df['z'] = df.x + df.y`
Standard groupby aggregations, joins, etc.	`result = df.groupby(df.z).y.mean()`
Compute result as a Pandas dataframe	`out = result.compute()`
Or store to CSV, Parquet, or other formats	`result.to_parquet('my-output.parquet')`

EXAMPLE	`df = dd.read_csv('filenames.*.csv')`
	`df.groupby(df.timestamp.day)\`
	`  .value.mean().compute()`

DASK ARRAYS	PARALLEL NUMPY ARRAYS FOR LARGE DATA
Import	`import dask.array as da`
Create from any array-like object	`import h5py` `dataset = h5py.File('my-data.hdf5')['/group/dataset']`
Including HFD5, NetCDF, or other on-disk formats.	`x = da.from_array(dataset, chunks=(1000, 1000))`
Alternatively generate an array from a random distribution.	`da.random.uniform(shape=(1e4, 1e4), chunks=(100, 100))`
Perform operations with NumPy syntax	`y = x.dot(x.T - 1) - x.mean(axis=0)`
Compute result as a NumPy array	`result = y.compute()`
Or store to HDF5, NetCDF or other on-disk format	`out = f.create_dataset(...)` `x.store(out)`

EXAMPLE	`with h5py.File('my-data.hdf5') as f:`
	`  x = da.from_array(f['/path'], chunks=(1000, 1000))`
	`  x -= x.mean(axis=0)`
	`  out = f.create_dataset(...)`
	`  x.store(out)`

DASK BAGS	PARELLEL LISTS FOR UNSTRUCTURED DATA
Import	`import dask.bag as db`
Create Dask Bag from a sequence	`b = db.from_sequence(seq, npartitions)`
Or read from text formats	`b = db.read_text('my-data.*.json')`

Map and filter results	`import json`
	`records = b.map(json.loads)`
	`  .filter(lambda d: d["name"] == "Alice")`

Compute aggregations like mean, count, sum	`records.pluck('key-name').mean().compute()`
Or store results back to text formats	`records.to_textfiles('output.*.json')`

EXAMPLE	`db.read_text('s3://bucket/my-data.*.json')`
	`  .map(json.loads)`
	`  .filter(lambda d: d["name"] == "Alice")`
	`  .to_textfiles('s3://bucket/output.*.json')`

ANACONDA.

CONTINUED ON BACK →

DASK COLLECTIONS (CONTINUED)

ADVANCED

Read from distributed file systems or cloud storage Prepend prefixes like hdfs://, s3://, or gcs:// to paths	`df = dd.read_parquet('s3://bucket/myfile.parquet')` `b = db.read_text('hdfs:///path/to/my-data.*.json')`
Persist lazy computations in memory	`df = df.persist()`
Compute multiple outputs at once	`dask.compute(x.min(), x.max())`

CUSTOM COMPUTATIONS — FOR CUSTOM CODE AND COMPLEX ALGORITHMS

DASK DELAYED — LAZY PARALLELISM FOR CUSTOM CODE

Import	`import dask`

Wrap custom functions with the @dask.delayed annotation	`@dask.delayed`
	`def load(filename):`
	`  ...`

Delayed functions operate lazily, producing a task graph rather than executing immediately	`@dask.delayed`
	`def process(data):`
	`  ...`

Passing delayed results to other delayed functions creates dependencies between tasks	`load = dask.delayed(load)` `process = dask.delayed(process)`
Call functions in normal code	`data = [load(fn) for fn in filenames]` `results = [process(d) for d in data]`
Compute results to execute in parallel	`dask.compute(results)`

CONCURRENT.FUTURES — ASYNCHRONOUS REAL-TIME PARALLELISM

Import	`from dask.distributed import Client`
Start local Dask Client	`client = Client()`
Submit individual task asynchronously	`future = client.submit(func, *args, **kwargs)`
Block and gather individual result	`result = future.result()`
Process results as they arrive	`for future in as_completed(futures):` `  ...`

EXAMPLE	`L = [client.submit(read, fn) for fn in filenames]`
	`L = [client.submit(process, future) for future in L]`
	`future = client.submit(sum, L)`
	`result = future.result()`

SET UP CLUSTER — HOW TO LAUNCH ON A CLUSTER

MANUALLY

Start scheduler on one machine	`$ dask-scheduler` `Scheduler started at SCHEDULER_ADDRESS:8786`
Start workers on other machines Provide address of the running scheduler	`host1$ dask-worker SCHEDULER_ADDRESS:8786` `host2$ dask-worker SCHEDULER_ADDRESS:8786`
Start Client from Python process	`from dask.distributed import Client` `client = Client('SCHEDULER_ADDRESS:8786')`

ON A SINGLE MACHINE

Call Client() with no arguments for easy setup on a single host	`client = Client()`

CLOUD DEPLOYMENT

See dask-kubernetes project for Google Cloud	`pip install dask-kubernetes`
See dask-ec2 project for Amazon EC2	`pip install dask-ec2`

MORE RESOURCES

User Documentation	dask.pydata.org
Technical documentation for distributed scheduler	distributed.readthedocs.org
Report a bug	github.com/dask/dask/issues

Python For Data Science *Cheat Sheet*

PySpark - SQL Basics

Learn Python for data science Interactively at www.DataCamp.com

PySpark & Spark SQL

Spark SQL is Apache Spark's module for working with structured data.

Initializing SparkSession

A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files.

```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession \
        .builder \
        .appName("Python Spark SQL basic example") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()
```

Creating DataFrames

From RDDs

```
>>> from pyspark.sql.types import *
```

Infer Schema

```
>>> sc = spark.sparkContext
>>> lines = sc.textFile("people.txt")
>>> parts = lines.map(lambda l: l.split(","))
>>> people = parts.map(lambda p: Row(name=p[0],age=int(p[1])))
>>> peopledf = spark.createDataFrame(people)
```

Specify Schema

```
>>> people = parts.map(lambda p: Row(name=p[0],
                                     age=int(p[1].strip())))
>>> schemaString = "name age"
>>> fields = [StructField(field_name, StringType(), True) for
field_name in schemaString.split()]
>>> schema = StructType(fields)
>>> spark.createDataFrame(people, schema).show()
  +--------+---+
  |    name|age|
  +--------+---+
  |    Mine| 28|
  |   Filip| 29|
  |Jonathan| 30|
  +--------+---+
```

From Spark Data Sources

JSON

```
>>> df = spark.read.json("customer.json")
>>> df.show()
 +--------------------+---+---------+--------+--------------------+
 |             address|age|firstName |lastName|         phoneNumber|
 +--------------------+---+---------+--------+--------------------+
 |[New York,10021,N...| 25|     John|   Smith|[[212 555-1234,ho...|
 |[New York,10021,N...| 21|     Jane|     Doe|[[322 888-1234,ho...|
 +--------------------+---+---------+--------+--------------------+
>>> df2 = spark.read.load("people.json", format="json")
```

Parquet files

```
>>> df3 = spark.read.load("users.parquet")
```

TXT files

```
>>> df4 = spark.read.text("people.txt")
```

Duplicate Values

```
>>> df = df.dropDuplicates()
```

Queries

```
>>> from pyspark.sql import functions as F
```

Select

```
>>> df.select("firstName").show()
>>> df.select("firstName","lastName") \
      .show()
>>> df.select("firstName",
              "age",
               explode("phoneNumber") \
              .alias("contactInfo")) \
      .select("contactInfo.type",
               "firstName",
               "age") \
      .show()
>>> df.select(df["firstName"],df["age"]+ 1)
      .show()
>>> df.select(df['age'] > 24).show()
```

When

```
>>> df.select("firstName",
              F.when(df.age > 30, 1) \
              .otherwise(0)) \
      .show()
>>> df[df.firstName.isin("Jane","Boris")]
                  .collect()
```

Like

```
>>> df.select("firstName",
              df.lastName.like("Smith")) \
      .show()
```

Startswith - Endswith

```
>>> df.select("firstName",
              df.lastName \
                .startswith("Sm")) \
      .show()
>>> df.select(df.lastName.endswith("th")) \
      .show()
```

Substring

```
>>> df.select(df.firstName.substr(1, 3) \
                         .alias("name")) \
      .collect()
```

Between

```
>>> df.select(df.age.between(22, 24)) \
      .show()
```

Show all entries in `firstName` column

Show all entries in `firstName`, `age` and `type`

Show all entries in `firstName` and `age`, add `1` to the entries of `age`
Show all entries where `age` >24

Show `firstName` and 0 or 1 depending on `age` >30

Show `firstName` if in the given options

Show `firstName`, and `lastName` is `TRUE` if `lastName` is like `Smith`

Show `firstName`, and `TRUE` if `lastName` starts with `Sm`

Show last names ending in `th`

Return substrings of `firstName`

Show `age`: values are `TRUE` if between `22` and `24`

Add, Update & Remove Columns

Adding Columns

```
>>> df = df.withColumn('city',df.address.city) \
           .withColumn('postalCode',df.address.postalCode) \
           .withColumn('state',df.address.state) \
           .withColumn('streetAddress',df.address.streetAddress) \
           .withColumn('telePhoneNumber',
                       explode(df.phoneNumber.number)) \
           .withColumn('telePhoneType',
                       explode(df.phoneNumber.type))
```

Updating Columns

```
>>> df = df.withColumnRenamed('telePhoneNumber', 'phoneNumber')
```

Removing Columns

```
>>> df = df.drop("address", "phoneNumber")
>>> df = df.drop(df.address).drop(df.phoneNumber)
```

GroupBy

```
>>> df.groupBy("age")\
      .count() \
      .show()
```

Group by `age`, count the members in the groups

Filter

```
>>> df.filter(df["age"]>24).show()
```

Filter entries of `age`, only keep those records of which the values are `>24`

Sort

```
>>> peopledf.sort(peopledf.age.desc()).collect()
>>> df.sort("age", ascending=False).collect()
>>> df.orderBy(["age","city"],ascending=[0,1])\
      .collect()
```

Missing & Replacing Values

```
>>> df.na.fill(50).show()
>>> df.na.drop().show()
>>> df.na \
      .replace(10, 20) \
      .show()
```

Replace null values
Return new `df` omitting rows with null values
Return new `df` replacing one value with another

Repartitioning

```
>>> df.repartition(10)\
      .rdd \
      .getNumPartitions()
>>> df.coalesce(1).rdd.getNumPartitions()
```

`df` with 10 partitions

`df` with 1 partition

Running SQL Queries Programmatically

Registering DataFrames as Views

```
>>> peopledf.createGlobalTempView("people")
>>> df.createTempView("customer")
>>> df.createOrReplaceTempView("customer")
```

Query Views

```
>>> df5 = spark.sql("SELECT * FROM customer").show()
>>> peopledf2 = spark.sql("SELECT * FROM global_temp.people")\
                 .show()
```

Output

Data Structures

```
>>> rdd1 = df.rdd
>>> df.toJSON().first()
>>> df.toPandas()
```

Convert `df` into an RDD
Convert `df` into a RDD of string
Return the contents of `df` as Pandas `DataFrame`

Write & Save to Files

```
>>> df.select("firstName", "city")\
      .write \
      .save("nameAndCity.parquet")
>>> df.select("firstName", "age") \
      .write \
      .save("namesAndAges.json",format="json")
```

Stopping SparkSession

```
>>> spark.stop()
```

Inspect Data

```
>>> df.dtypes
>>> df.show()
>>> df.head()
>>> df.first()
>>> df.take(2)
>>> df.schema
```

Return `df` column names and data types
Display the content of `df`
Return first `n` rows
Return first row
Return the first `n` rows
Return the schema of `df`

```
>>> df.describe().show()
>>> df.columns
>>> df.count()
>>> df.distinct().count()
>>> df.printSchema()
>>> df.explain()
```

Compute summary statistics
Return the columns of `df`
Count the number of rows in `df`
Count the number of distinct rows in `df`
Print the schema of `df`
Print the (logical and physical) plans

Python For Data Science *Cheat Sheet*

PySpark - RDD Basics

Learn Python for data science Interactively at www.DataCamp.com

Spark

PySpark is the Spark Python API that exposes the Spark programming model to Python.

Initializing Spark

SparkContext

```
>>> from pyspark import SparkContext
>>> sc = SparkContext(master = 'local[2]')
```

Inspect SparkContext

```
>>> sc.version                  Retrieve SparkContext version
>>> sc.pythonVer                Retrieve Python version
>>> sc.master                   Master URL to connect to
>>> str(sc.sparkHome)           Path where Spark is installed on worker nodes
>>> str(sc.sparkUser())         Retrieve name of the Spark User running SparkContext
>>> sc.appName                  Return application name
>>> sc.applicationId            Retrieve application ID
>>> sc.defaultParallelism       Return default level of parallelism
>>> sc.defaultMinPartitions     Default minimum number of partitions for RDDs
```

Configuration

```
>>> from pyspark import SparkConf, SparkContext
>>> conf = (SparkConf()
            .setMaster("local")
            .setAppName("My app")
            .set("spark.executor.memory", "1g"))
>>> sc = SparkContext(conf = conf)
```

Using The Shell

In the PySpark shell, a special interpreter-aware SparkContext is already created in the variable called `sc`.

```
$ ./bin/spark-shell --master local[2]
$ ./bin/pyspark --master local[4] --py-files code.py
```

Set which master the context connects to with the `--master` argument, and add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to `--py-files`.

Loading Data

Parallelized Collections

```
>>> rdd = sc.parallelize([('a',7),('a',2),('b',2)])
>>> rdd2 = sc.parallelize([('a',2),('d',1),('b',1)])
>>> rdd3 = sc.parallelize(range(100))
>>> rdd4 = sc.parallelize([("a",["x","y","z"]),
                           ("b",["p", "r"])])
```

External Data

Read either one text file from HDFS, a local file system or or any Hadoop-supported file system URI with `textFile()`, or read in a directory of text files with `wholeTextFiles()`.

```
>>> textFile = sc.textFile("/my/directory/*.txt")
>>> textFile2 = sc.wholeTextFiles("/my/directory/")
```

Retrieving RDD Information

Basic Information

```
>>> rdd.getNumPartitions()      List the number of partitions
>>> rdd.count()                 Count RDD instances
 3
>>> rdd.countByKey()            Count RDD instances by key
 defaultdict(<type 'int'>,{'a':2,'b':1})
>>> rdd.countByValue()          Count RDD instances by value
 defaultdict(<type 'int'>,{('b',2):1,('a',2):1,('a',7):1})
>>> rdd.collectAsMap()          Return (key,value) pairs as a dictionary
 {'a': 2,'b': 2}
>>> rdd3.sum()                  Sum of RDD elements
 4950
>>> sc.parallelize([]).isEmpty()  Check whether RDD is empty
 True
```

Summary

```
>>> rdd3.max()                  Maximum value of RDD elements
  99
>>> rdd3.min()                  Minimum value of RDD elements
  0
>>> rdd3.mean()                 Mean value of RDD elements
  49.5
>>> rdd3.stdev()                Standard deviation of RDD elements
  28.866070047722118
>>> rdd3.variance()             Compute variance of RDD elements
  833.25
>>> rdd3.histogram(3)           Compute histogram by bins
  ([0,33,66,99],[33,33,34])
>>> rdd3.stats()                Summary statistics (count, mean, stdev, max & min)
```

Applying Functions

```
>>> rdd.map(lambda x: x+(x[1],x[0]))   Apply a function to each RDD element
       .collect()
  [('a',7,7,'a'),('a',2,2,'a'),('b',2,2,'b')]
>>> rdd5 = rdd.flatMap(lambda x: x+(x[1],x[0]))   Apply a function to each RDD element and flatten the result

>>> rdd5.collect()
  ['a',7,7,'a','a',2,2,'a','b',2,2,'b']
>>> rdd4.flatMapValues(lambda x: x)   Apply a flatMap function to each (key,value) pair of rdd4 without changing the keys
       .collect()
  [('a','x'),('a','y'),('a','z'),('b','p'),('b','r')]
```

Selecting Data

Getting

```
>>> rdd.collect()               Return a list with all RDD elements
  [('a', 7), ('a', 2), ('b', 2)]
>>> rdd.take(2)                 Take first 2 RDD elements
  [('a', 7), ('a', 2)]
>>> rdd.first()                 Take first RDD element
  ('a', 7)
>>> rdd.top(2)                  Take top 2 RDD elements
  [('b', 2), ('a', 7)]
```

Sampling

```
>>> rdd3.sample(False, 0.15, 81).collect()   Return sampled subset of rdd3
  [3,4,27,31,40,41,42,43,60,76,79,80,86,97]
```

Filtering

```
>>> rdd.filter(lambda x: "a" in x)   Filter the RDD
       .collect()
  [('a',7),('a',2)]
>>> rdd5.distinct().collect()   Return distinct RDD values
  ['a',2,'b',7]
>>> rdd.keys().collect()        Return (key,value) RDD's keys
  ['a', 'a', 'b']
```

Iterating

```
>>> def g(x): print(x)
>>> rdd.foreach(g)              Apply a function to all RDD elements
  ('a', 7)
  ('b', 2)
  ('a', 2)
```

Reshaping Data

Reducing

```
>>> rdd.reduceByKey(lambda x,y : x+y)   Merge the rdd values for each key
       .collect()
  [('a',9),('b',2)]
>>> rdd.reduce(lambda a, b: a + b)   Merge the rdd values
  ('a',7,'a',2,'b',2)
```

Grouping by

```
>>> rdd3.groupBy(lambda x: x % 2)   Return RDD of grouped values
        .mapValues(list)
        .collect()
>>> rdd.groupByKey()            Group rdd by key
       .mapValues(list)
       .collect()
  [('a',[7,2]),('b',[2])]
```

Aggregating

```
>>> seqOp = (lambda x,y: (x[0]+y,x[1]+1))
>>> combOp = (lambda x,y:(x[0]+y[0],x[1]+y[1]))
>>> rdd3.aggregate((0,0),seqOp,combOp)   Aggregate RDD elements of each partition and then the results
  (4950,100)
>>> rdd.aggregateByKey((0,0),seqop,combop)   Aggregate values of each RDD key
       .collect()
  [('a',(9,2)), ('b',(2,1))]
>>> rdd3.fold(0,add)            Aggregate the elements of each partition, and then the results
  4950
>>> rdd.foldByKey(0, add)       Merge the values for each key
       .collect()
  [('a',9),('b',2)]
>>> rdd3.keyBy(lambda x: x+x)   Create tuples of RDD elements by applying a function
       .collect()
```

Mathematical Operations

```
>>> rdd.subtract(rdd2)          Return each rdd value not contained in rdd2
       .collect()
  [('b',2),('a',7)]
>>> rdd2.subtractByKey(rdd)     Return each (key,value) pair of rdd2 with no matching key in rdd
       .collect()
  [('d', 1)]
>>> rdd.cartesian(rdd2).collect()   Return the Cartesian product of rdd and rdd2
```

Sort

```
>>> rdd2.sortBy(lambda x: x[1])   Sort RDD by given function
       .collect()
  [('d',1),('b',1),('a',2)]
>>> rdd2.sortByKey()            Sort (key, value) RDD by key
       .collect()
  [('a',2),('b',1),('d',1)]
```

Repartitioning

```
>>> rdd.repartition(4)          New RDD with 4 partitions
>>> rdd.coalesce(1)             Decrease the number of partitions in the RDD to 1
```

Saving

```
>>> rdd.saveAsTextFile("rdd.txt")
>>> rdd.saveAsHadoopFile("hdfs://namenodehost/parent/child",
                         'org.apache.hadoop.mapred.TextOutputFormat')
```

Stopping SparkContext

```
>>> sc.stop()
```

Execution

```
$ ./bin/spark-submit examples/src/main/python/pi.py
```