Python Workshop

Jan 14, 2021

Variable

Declaration of variables is not required in Python.

- integer
  - The size of interger could be unlimited. No need to worry about integer overflow.

```
>>> i = 2**100
>>> i
1267650600228229401496703205376
```

  - Support multiple bases

```
>>> i, j, k = 0b100, 0o11, 0xAA
>>> print(i,j,k)
4 9 170
```

Variable

- float
- boolean
  - True, False
- complex number

```
>>> import math, cmath
>>> x = math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> x = cmath.sqrt(-1)
>>> type(x)
<class 'complex'>
>>> x = 2 + 3j
>>> x
(2+3j)
```

Operation

- Addition/Substration/Multiplication - same as Java/C
- Division
  - python 2: 1/2 = 0
  - python 3: 1/2 = 0.5
- Floor/Integer Division
  - python 3: 1//2 = 0, -23//10 = -3 (round to floor)
  - Java/C: -23/10 = -2 (round to zero)
- Modulus
  - python 3: -23%10 = 7 (-23 = -3 * 10 + 7)
  - Java/C: -23%10 = -3 (-23 = -2 * 10 - 3)

Assignment

Practice:

Write a piece of pseudocode to swap values of two variables a and b.

Assignment

Practice:

Write a piece of pseudocode to swap values of two variables a and b.

Java/C solution:

```
tmp = a
a = b
b = tmp
```

Assignment

Practice:

Write a piece of pseudocode to swap values of two variables a and b.

Java/C solution:

```
tmp = a
a = b
b = tmp
```

Python solution:

```
a,b = b,a
```

Assignment

Another Example

Given a singly linked list, insert a node *p* after current node *cur*.

Java/C solution:

```
tmp = cur.next
cur.next = p
p.next = tmp
```

In python, we only need a one-line code to insert *p* after *cur*.

Assignment

Another Example

Given a singly linked list, insert a node *p* after current node *cur*.

Java/C solution:

```
tmp = cur.next
cur.next = p
p.next = tmp
```

In python, we only need a one-line code to insert *p* after *cur*.

```
cur.next, p.next = p, cur.next
```

Typecasting/Conversion

- int()

```
>>> int(1.9), int('100'), int('100', base=2)
(1, 100, 4)
```

- float()

```
>>> float(2), float('1.234')
(2.0, 1.234)
```

- str()

```
>>> str(3.1415)
'3.1415'
```

List

- Initialization/Generation

```
>>> [1,2,3] # one dimension
[1, 2, 3]
>>> [[1,2], [3,4]] # two dimensions, list of list
[[1, 2], [3, 4]]
>>> [1.0, 2, 'str'] # each element can be any type of object
[1.0, 2, 'str']
```

range(start, end, stride)

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1,5))
[1, 2, 3, 4]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
```

List

- Initialization/Generation

```
>>> [1, 2, 3] + [4, 5] # concatenation
[1, 2, 3, 4, 5]

>>> [2] * 3 # duplication
[2, 2, 2]

>>> [1,2,3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List

- Slicing
  - index:
    - [0, 1, 2, …. , len(list)-2, len(list)-1]
    - [0, 1, 2, …. , -2, -1]

```
>>> x = list(range(10))
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[0], x[2], x[-1], x[-3]
(0, 2, 9, 7)
```

List

- Slicing
  - [start : end: stride]
    - default stride is 1
    - if stride > 0:
      - for(i = start; i < end; i += stride)
      - default start is the first one
      - default end is the last one

List

- Slicing if stride > 0

```
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[1:4] # 1:4:1, defaut stride is 1
[1, 2, 3]

>>> x[:4] # 0:4:1, default start is the first one
[0, 1, 2, 3]

>>> x[4:] # 4:11:1, default end is the last one
[4, 5, 6, 7, 8, 9]
```

List

- Slicing
  - [start : end: stride]
    - if stride < 0:
      - for(i = start; i > end; i += stride)
      - default start is the last one
      - default end is the first one

List

- Slicing if stride < 0

```
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[6:1:-1]
[6, 5, 4, 3, 2]
>>> x[6::-1] # default end is the first one
[6, 5, 4, 3, 2, 1, 0]
>>> x[:6:-1]# 9:6:-1 or -1:-4:-1, default start is the last one
[9, 8, 7]
```

Question

Can we reverse a list through slicing? How?

List

- Slicing

  Question
  Can we reverse a list through slicing? How?

```
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[::-1] # -1:-11:-1
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

List

- Method
  - len: `len(x)`
  - append: `x.append(3)`
  - pop: `x.pop(), x.pop(0)`
  - insert: `x.insert(0,7)`
  - count: `x.count(3)`
  - index: `x.index(3)`
  - sort `x.sort()`

Question

Can we use a list as a stack or a queue? How?

List

- Method

  Question
  Can we use a list as a stack or a queue? How?

  - stack: LIFO
    - push: append(e)
    - pop: pop()
  - queue: FIFO
    - add/remove: append(e) / pop(0)
    - add/remove: insert(0, e) / pop()

List

- mutable

```
>>> x = [1,2,3]
>>> x[0] = 0
>>> x
[0, 2, 3]
```

pass by reference

```
>>> def change(x): x[0]=0

>>> x=[1,2,3]
>>> change(x)
>>> x
[0, 2, 3]
```

List

- copy

  Dangerous Zone

```
>>> x = [1,2,3]
>>> y = x
>>> x[0] = 0
>>> x
[0, 2, 3]
>>> y
????
```

List

- copy

  Dangerous Zone

```
>>> x = [1,2,3]
>>> y = x
>>> x[0] = 0
>>> x
[0, 2, 3]
>>> y
[0, 2, 3] # copy by reference
```

  - copy by value

```
>>> x = [1,2,3]
>>> y = x[:]
>>> y = x.copy()
>>> y = list(x)
```

List

- copy

  Dangerous Zone

```
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> x[0][0] = 0
>>> x
[[0, 2], [3, 4]]
>>> y
???
```

List

- copy

  Dangerous Zone

```
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> x[0][0] = 0
>>> x
[[0, 2], [3, 4]]
>>> y
[[0, 2], [3, 4]] # shallow copy
```

  - deep copy

```
>>> from copy import deepcopy as copy
>>> y = copy(x)
```

List

- comprehension

```
myList = []
for i in range(10):
     myList.append(i)

>>> myList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

is equivalent to

```
>>> myList = [i for i in range(10)]

>>> myList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List

- comprehension

`if condition`

```
myList = []
for i in range(10):
    if i%2:
        myList.append(i)
```

is equivalent to

```
>>> myList = [i for i in range(10) if i%2]

>>> myList
[1, 3, 5, 7, 9]
```

List

- comprehension

`nested for loop in one line ...`

```
myList = []
for i in range(5):
    for j in range(i):
        myList.append(i+j)
```

is equivalent to

```
>>> myList = [i+j for i in range(5) for j in range(i)]

>>> myList
[1, 2, 3, 3, 4, 5, 4, 5, 6, 7]
```

List

- comprehension

  `Application:`

  create a 3 by 4 list with all 0

```
>>> x = [[0 for i in range(3)] for j in range(4)]
>>> x
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

  Do not use this to create two dimentional list:

```
>>> x = [[0] * 3] * 4
```

List

- comprehension

  Practice
  Use list comprehension to convert a string to a list like this:

  convert "hello" to ['h', 'e', 'l', 'l', 'o'].

List

- comprehension

  Practice
  Use list comprehension to convert a string to a list like this:

  convert "hello" to ['h', 'e', 'l', 'l', 'o'].

```
>>> s = 'hello'
>>> [i for i in s]
['h', 'e', 'l', 'l', 'o']
```

String

- immutable

```
>>> s = 'hello'
>>> s[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- slicing: same as list

```
>>> s[::-1]
'olleh'
```

- concatenation

```
>>> 'Westminster' + 'College' + str(100)
'WestminsterCollege100'
```

String

- method
  - str.strip(), str.split(pattern), pattern.join(str)

```
>>> path = '  doc/cmpt/360/final  '

>>> path = path.strip()
>>> path
'doc/cmpt/360/final'

>>> path = path.split('/')
>>> path
['doc', 'cmpt', '360', 'final']

>>> path = '/'.join(path)
>>> path
'doc/cmpt/360/final'
```

tuple

- immutable
  tuple is similar with list, but it is immutable.

```
>>> t = (1,2,3)
>>> t[0] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- create a tupe with only one element

```
  >>> (1) # Wrong.
  1
  >>> (1,) # correct
  (1,)
  >>> 1, # correct
  (1,)
```

tuple

- pack and unpack

```
>>> t = 3,4 # packing
>>> t
(3, 4)
>>> a,b = t # unpacking
>>> a
3
>>> b
4
```

- concatenation

```
>>> t + (5,6)
(3, 4, 5, 6)

>>> t + tuple([7,8])
(3, 4, 7, 8)
```

dictionary

- create

  dictionary (hash tables) is mutable.

```
>>> d = {} # empty dictionary
>>> d['x'] = 3 # add one item
>>> d['y'] = 5
>>> d['x'] = 4 # change the value of key x
>>> d
{'x': 4, 'y': 5}

>>> d2 = {'x':7, 'y':8} # key:value
>>> d2
{'x': 7, 'y': 8}
```

dictionary

- key, value

```
>>> len(d) # number of keys
2

>>> list(d.keys())
['x', 'y']
>>> list(d.values())
[7, 8]
>>> list(d.items())
[('x', 7), ('y', 8)]

>>> 'y' in d # check if key y in d
True

>>> for key in d: print(key, d[key]) # loop over d
x 7
y 8
```

set

- create

  No duplication. No order.

```
>>> s={1,2,3,4,4,4,4}
>>> s
{1, 2, 3, 4}

>>> s = set([1,2,3,4,4,4]) # create a set from a list
>>> s
{1, 2, 3, 4}
```

Application
Given a list, can we use set to remove all duplications from this list?

set

- create

  Application
  Given a list, can we use set to remove all duplications from this list?

```
>>> a = [1,2,2,3,4,2,1,5]
>>> list(set(a))
[1, 2, 3, 4, 5]
```

set

- operation

```
>>> s1 = {1,2,3,4}
>>> s2= {3,4,5}

>>> s1 & s2 # intersection, and
{3, 4}
>>> s1 | s2 # union, or
{1, 2, 3, 4, 5}
>>> s1 ^ s2 # XOR, exclusive or
{1, 2, 5}
>>> s1 - s2 # difference
{1, 2}
>>> s2 - s1 # difference
{5}
```

built-in function

- `format`: string formatting

```
>>> 'score: ' + str(90) + '/' + str(100)
'score: 90/100'

# use format
>>> 'score: {}/{}'.format(90,100)
'score: 90/100'
```

More examples can be found from https://pyformat.info/

built-in function

- `zip`: aggregrate two iterable objects.

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> zip(a,b)
<zip object at 0x10254bc48>
>>> list(zip(a,b))
[(1, 4), (2, 5), (3, 6)]
```

- `enumerate`: add counter to iterable objects

```
>>> a=[4,5,6]
>>> enumerate(a)
<enumerate object at 0x10254e900>
>>> list(enumerate(a))
[(0, 4), (1, 5), (2, 6)]
```

built-in function

- list comprehension and zip

  Practice
  Calculate sum of the element product of two arrays a and b.

built-in function

- list comprehension and zip

  Practice
  Calculate sum of the element product of two arrays a and b.

  Normal way

```
wsum=0
for i,j in zip(a,b):
    wsum += i*j
```

  Another way

```
wsum = sum([i*j for i,j in zip(a,b)])
```

lambda

- create anonymous functions

```
def by_three(x):
    return x%3 == 0

>>> by_three(6)
True
```

is same as

```
>>> by_three = lambda x: x % 3 == 0
>>> by_three(6)
True
```

lambda

- style of functional programming

```
def add_number(n):
    return lambda x: x+n

>>> add_three = add_number(3)
>>> add_three(4)
7

>>> add_five = add_number(5)
>>> add_five(4)
9
```

lambda with map, filter, reduce

- `map`: map a function onto each element of an iterable object
  - Example 1: convert each element in a list to a string

```
a = [1,2,3,4]
>>> list(map(str, a))
['1', '2', '3', '4']
```

  - Example 2: double each element in a list by 2

```
a = [1,2,3,4]
>>> list(map(lambda x: x*2, a))
[2, 4, 6, 8]
```

lambda with map, filter, reduce

- `filter`: filter out all elements under a condition
  - Example 1: remove all even numbers from a list

```
a = [1,2,3,4]
>>> list(filter(lambda x: x%2, a))
[1, 3]
```

  - Example 2: remove all numbers less than 3 from a list

```
a = [1,2,3,4]
>>> list(filter(lambda x: x>=3, a))
[3, 4]
```

lambda with map, filter, reduce

- `reduce`: reduce is equivalent to cumulatively apply function to two inputs in a list
  - Example: calculate product of a list
    - normal way

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num
```

    - use reduce

```
>>> from functools import reduce
>>> from operator import mul
>>> product = reduce(mul, [1, 2, 3, 4])
>>> product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
```

function

- Python function can return multiple values as a tuple.

```
def f(a = 0): # default value of a is 0
    return a, a+1, a+2

>>> a, b, c = f(3)
```

- pass statement: placeholder for future implementation.

```
def future():
    pass
```

function

- scope:
  - global

```
a = 0
def my_function():
    print(a) # print global a

my_function()
```

  - local

```
a = 0
def my_function():
    a = 3 # create a local aï¼Œwhich always takes precedence
    print(a) # print local a

my_function()
print(a) # print global a
```

function

- scope

  Is this correct?

```
a = 0

def my_function():
    print(a)
    a = 3
    print(a)

my_function()
```

function

- scope

  Is this correct?

```
a = 0

def my_function():
    # Should refer to local a, but a has not been defined.
    print(a)
    # Create a local a.
    # We cannot refer global a elsewhere in this function.
    a = 3
    print(a)

my_function()

>>> UnboundLocalError: local variable 'a' referenced
    before assignment
```

for loop

Java/C++ style:

```
for(int i = 0; i < 100: i ++){
    cout<< list[i] << endl;
}
```

python style:

```
for index in range(len(list):
    print(list[index])
```

or

```
for element in list:
    print(element)
```

if condition

```
if cond1 and cond2:
    statement
elif not cond3:
    statement
else:
    statement
```

Check if a list is empty

```
if len(list) != 0:
    do something
```

A better way:

```
if list:
   do something
```

if condition

Check if x is in path

```
for i in path:
    if i == 'x':
        do something
```

A better way:

```
if 'x' in path:
   do something
```

class

```
class Animal():
    is_alive = True    # member variable

    # construction method.
    # self is like this pointer
    def __init__(self, name, age):
        local_variable = 4 # local variable
        self.name = name # instance variable
        self.age = age
        self.sides = 4 # instance variable without inputing value

     def my_function(self, var):
        self.my_function(var) # always use self.
```

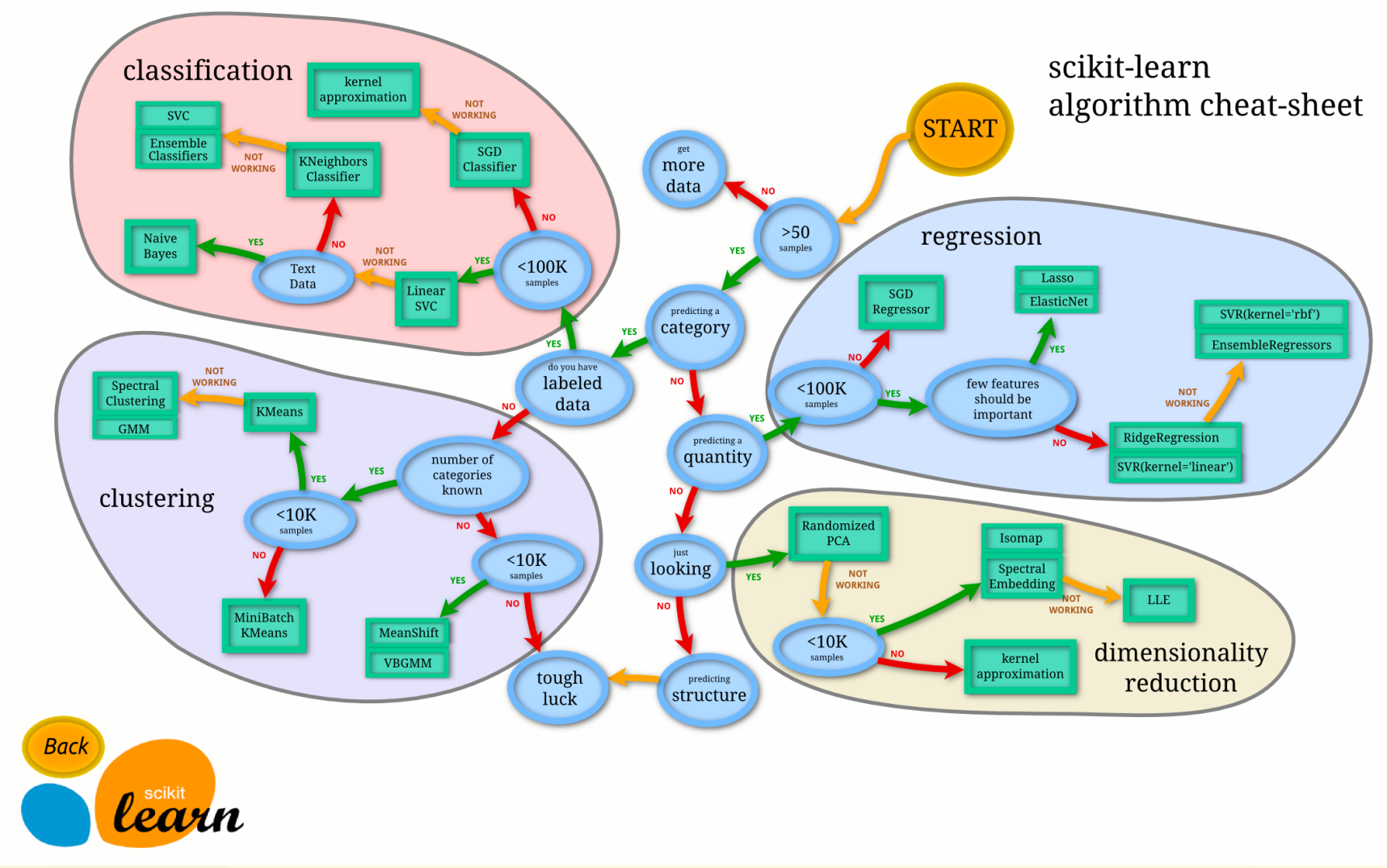
library

- math: `math`
- priority queue: `heapq`
- permutation, combinations, etc: `itertools`
- regular expression: `re`
- parser for command-line: `argparse`
- system command, path, etc: `sys, os`
- image processing: `pillow`
- web development: `django, flask`
- sql: `sqlite3`
- .....

data science

- array, matrix: `numpy`
- linear algebra, statistics: `scipy`
- symbolic calculation: `sympy`
- visualization: `matplotlib, seaborn`
- data manipulation and analysis: `pandas`
- machine learning: `scikit-learn`
- deep learning: `TensorFlow, PyTorch`

sklearn


scikit-learn
algorithm cheat-sheet
START
>50 samples
NO
get more data
YES
predicting a category
YES
do you have labeled data
NO
predicting a quantity
YES
<100K samples
NO
SGD Regressor
YES
few features should be important
YES
Lasso
ElasticNet
NO
RidgeRegression
SVR(kernel='linear')
NOT WORKING
SVR(kernel='rbf')
EnsembleRegressors
regression
NO
just looking
YES
Randomized PCA
NOT WORKING
<10K samples
YES
Isomap
Spectral Embedding
NOT WORKING
LLE
NO
kernel approximation
dimensionality reduction
NO
predicting structure
tough luck
NO
number of categories known
YES
<10K samples
YES
KMeans
NOT WORKING
Spectral Clustering
GMM
NO
MiniBatch KMeans
<10K samples
YES
MeanShift
VBGMM
clustering
<100K samples
YES
kernel approximation
NOT WORKING
SGD Classifier
NO
KNeighbors Classifier
NOT WORKING
SVC
Ensemble Classifiers
NO
Linear SVC
NOT WORKING
Text Data
YES
Naive Bayes
YES
classification
Back
scikit learn

numpy

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.

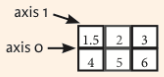
Use the following import convention:

```
>>> import numpy as np
```

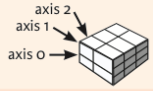
NumPy Arrays

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]                             Select the element at the 2nd index
  3
>>> b[1,2]                           Select the element at row 0 column 2
  6.0                                (equivalent to b[1][2])
```

Slicing

```
>>> a[0:2]                           Select items at index 0 and 1
  array([1, 2])
>>> b[0:2,1]                         Select items at rows 0 and 1 in column 1
  array([ 2.,  5.])
>>> b[:1]                            Select all items at row 0
  array([[1.5, 2., 3.]])             (equivalent to b[0:1, :])
>>> c[1,...]                         Same as [1,:,:]
  array([[[ 3.,  2.,  1.],
          [ 4.,  5.,  6.]]])
>>> a[ : :-1]                        Reversed array a
  array([3, 2, 1])
```

Boolean Indexing

```
>>> a[a<2]                           Select elements from a less than 2
  array([1])
```

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]    Select elements (1,0),(0,1),(1,2) and (0,0)
  array([ 4. ,  2. ,  6. ,  1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]    Select a subset of the matrix's rows
  array([[ 4. ,  5. ,  6. ,  4. ],   and columns
         [ 1.5,  2. ,  3. ,  1.5],
         [ 4. ,  5. ,  6. ,  4. ],
         [ 1.5,  2. ,  3. ,  1.5]])
```

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
  [array([1]),array([2]),array([3])]  index
>>> 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.]]])]
```

DataCamp
Learn Python for Data Science Interactively

More Resources:

- More cheatsheet: https://github.com/kailashahirwar/cheatsheets-ai
- Foundations of Python Programming https://runestone.academy/runestone/static/fopp/index.html
- Python 3 tutorial: https://www.python-course.eu/python3_course.php
- Think Python: http://greenteapress.com/thinkpython2/thinkpython2.pdf

Thanks