LAB 1:
PYTHON FOUNDATION

University of Washington, Seattle

Spring 2022

OUTLINE

Part 1: Getting Started (video link)

- Python Environment Setup

Part 2: Python Basics (video link)

- Data types & variables
- Operators in Python
- Conditionals, Loops, Functions

Part 3: NumPy and Plotting (video link)

- Introduction to NumPy
- Plotting with Matplotlib

Supplementary: Basic Debugging in Python (video link)

- Tips for minimizing errors
- Debugging with 'print'
- Debugging with PDB
- Using Google / Stack Overflow

PART 1:

GETTING STARTED

Python Environment Setup

Python Environment Setup

Python Environment Options

Anaconda 3

Offline

Google Colaboratory

Online

What is Anaconda 3?

Distribution of Python and R for scientific computing

- Comes with >250 packages automatically installed
- >7500 additional open-source packages available in conda website
- Equipped with Jupyter Notebook
- Conda environment manager for easy maintenance of packages

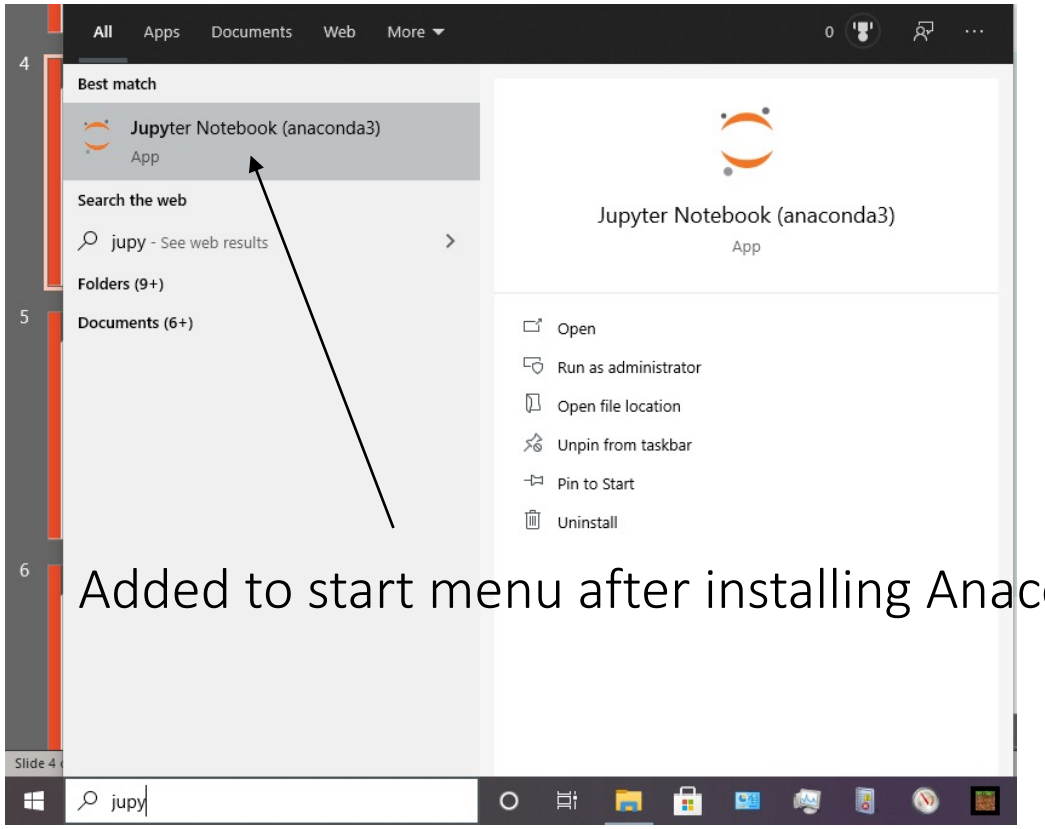
Setting up Anaconda 3

Installing Anaconda 3

https://www.anaconda.com/products/individual

Setting up Anaconda 3

Windows



Added to start menu after installing Anaconda 3

Mac/Linux

Start terminal

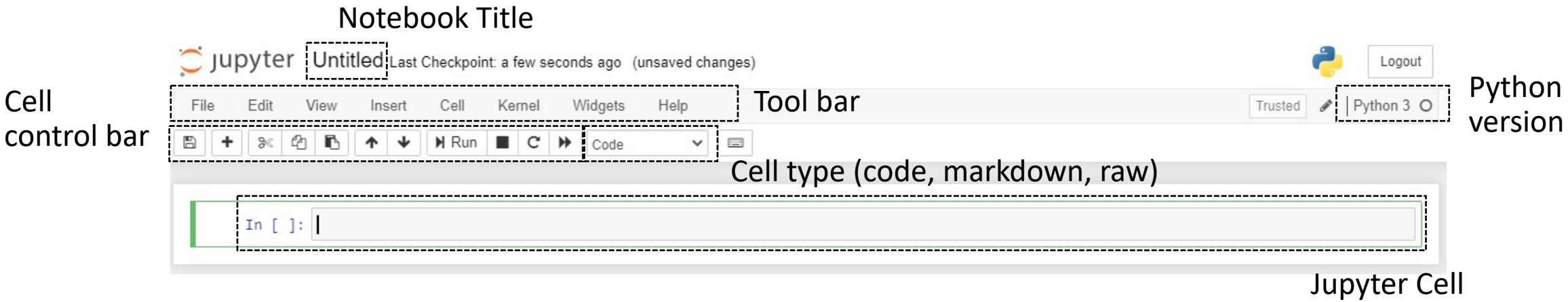
Type “jupyter notebook”

Setting up Anaconda 3

Create a new notebook

You can also use Jupyter Navigator to load .ipynb notebook files

Jupyter Notebook



See https://www.dataquest.io/blog/jupyter-notebook-tutorial to familiarize yourself with basic controls

Google Colaboratory

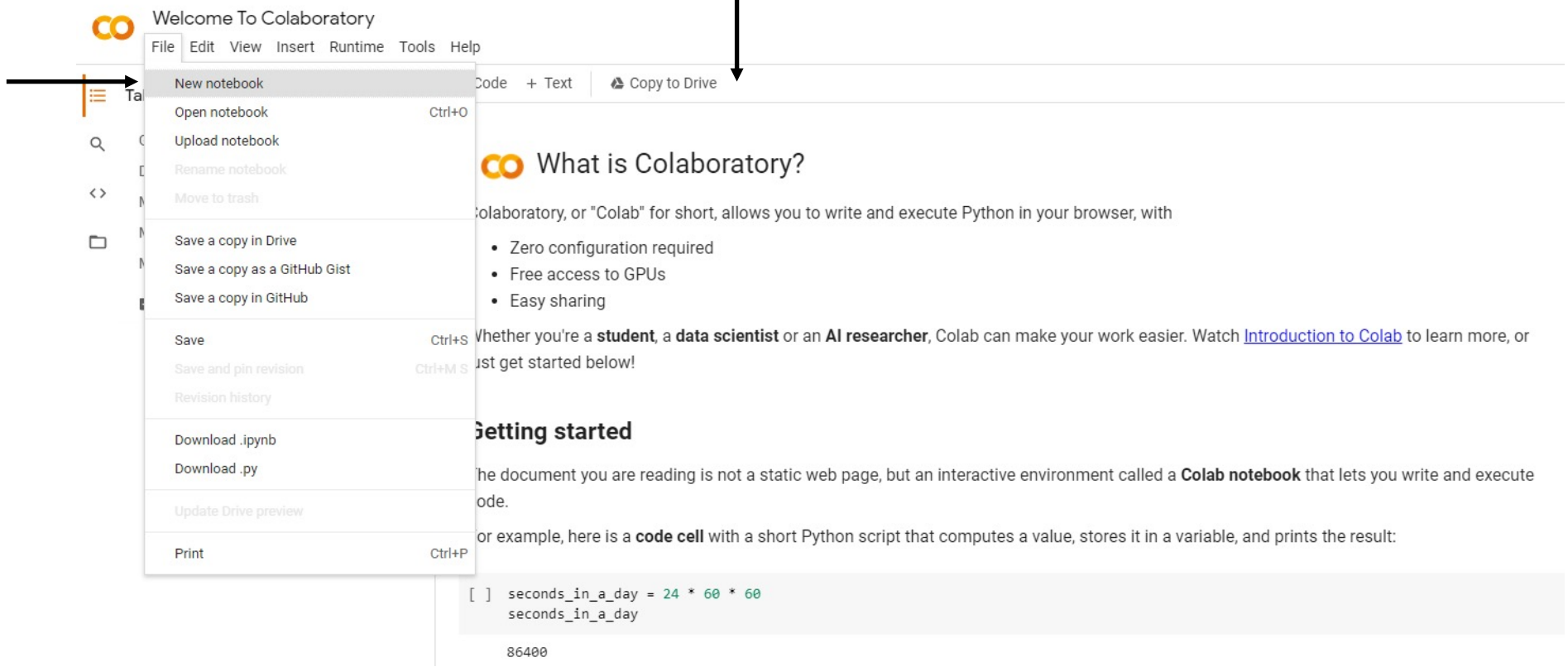
A free Jupyter notebook environment that runs in the cloud

- Saves in Google drive
- Github commit style code sharing with others
- Maximum runtime of 12hrs (Free version)
- Pre-equipped with latest scientific packages (Numpy, Scipy, etc)

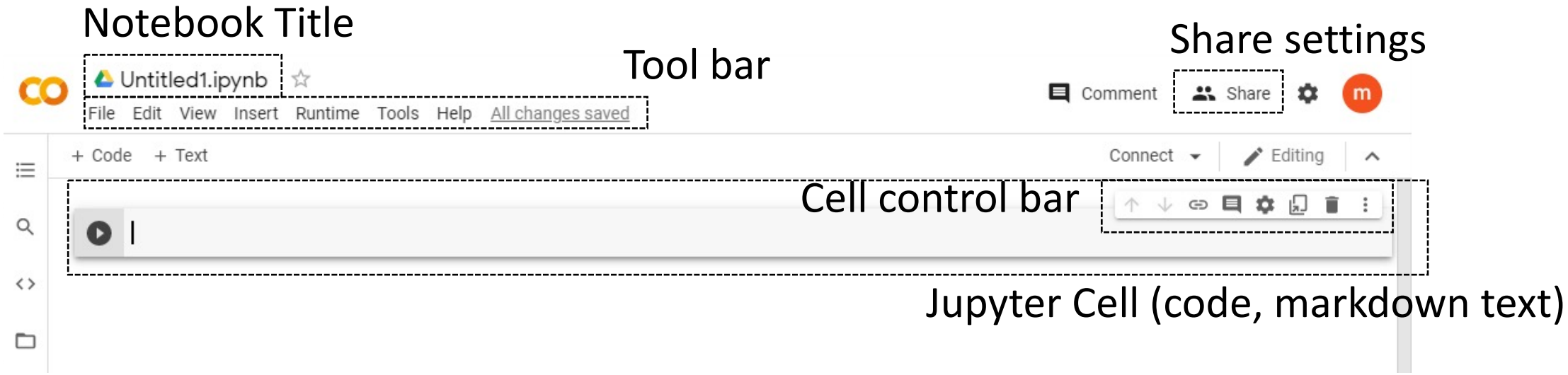
Setting up Google Colaboratory

Tutorial to Colab https://colab.research.google.com/notebooks/intro.ipynb

Create new Notebook



Setting up Google Colaboratory

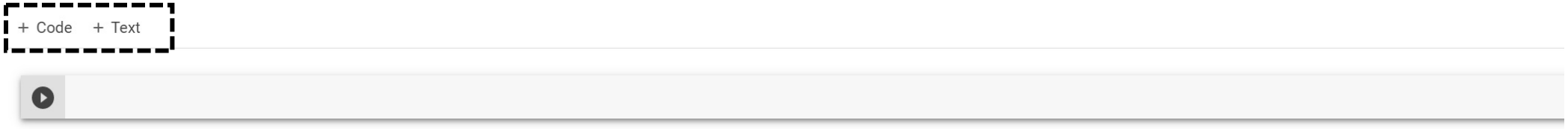


See **Getting Started** part of https://colab.research.google.com/notebooks/intro.ipynb to familiarize yourself with basic controls

Code vs Markdown Cell

Jupyter Notebook

Google Colab



PART 2:

PYTHON BASICS

Python Data Types and Variables

Operators in Python

Conditionals, Loops, Functions

Python Data Types and Variables

Python Data Types and Variables

Jupyter Notebook Code

```
In [1]: x = 1
        print(x)

        1

In [2]: y = 2.5
        print(y)

        2.5

In [3]: z = True
        print(z)

        True

In [4]: s = 'hello'
        print(s)

        hello
```

Variable	Data Type	Value
x	int	1
y	float	2.5
z	bool	True
s	str	‘hello’

Printing Variables with ‘print’

Print single variable

```
var1 = 2021
var2 = 'Fall'

print(var1)
```

```
2021
```

Print multiple variable

```
print(var1, var2)
```

```
2021 Fall
```

Variables called in a cell can be displayed without print function, as ‘outputs’

```
var1
```

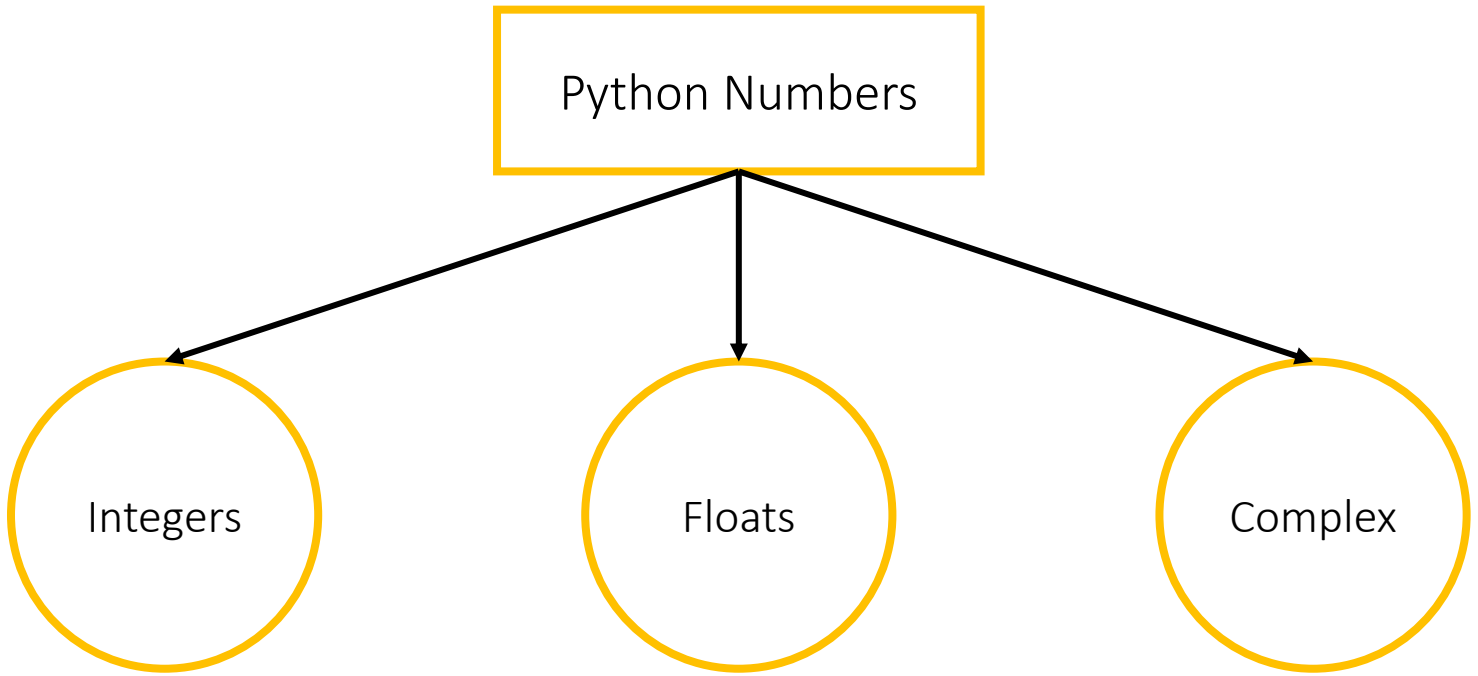
```
2021
```

```
var1, var2
```

```
(2021, 'Fall')
```

Python Data Types: Numbers

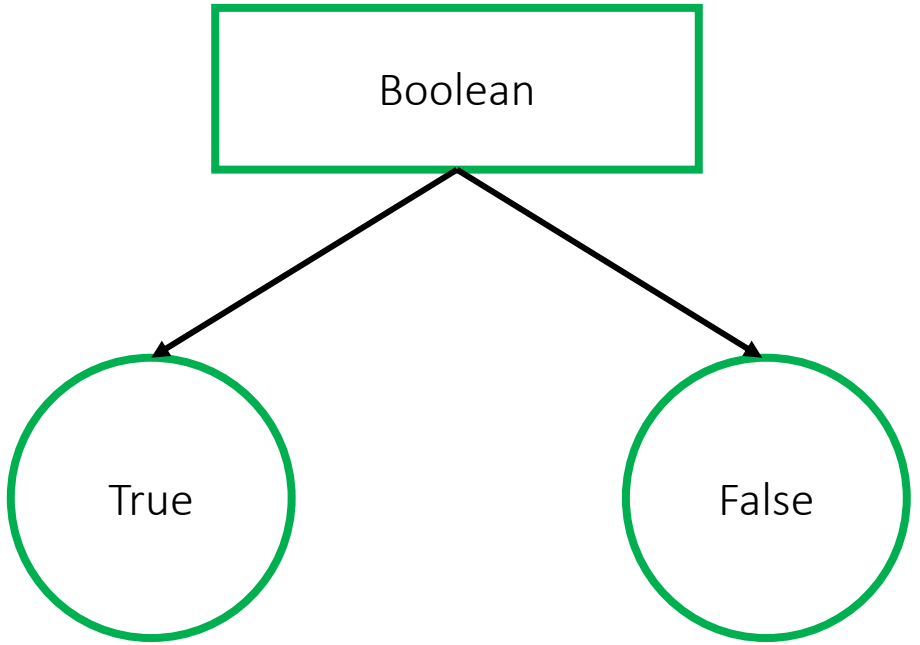
```
x = 1        # Integer
y = 1.6      # Float
z = 2 + 6j   # Complex
```



Examples	1, 300, -50	0.001, -15.7, 6.5, 1.0	6.4 – 3.6j, 8j, -10-2j

Python Data Types: Booleans

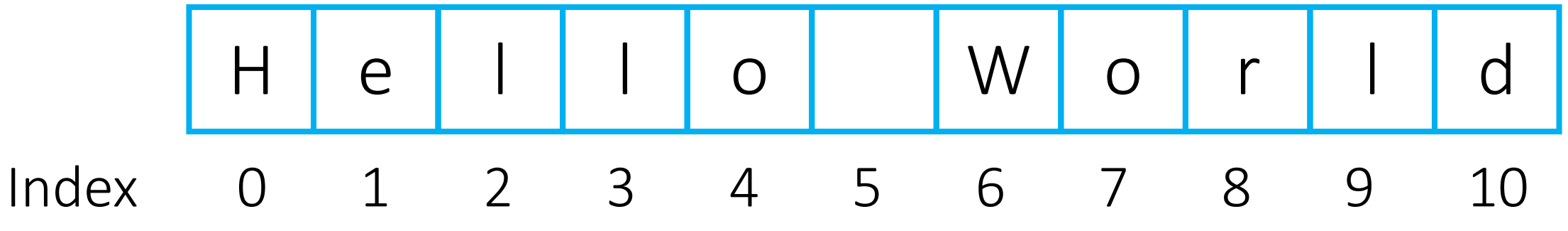
```
x = True      # True

y = False     # False
```



First letter should be capitalized

Python Data Type: Strings

```
x = 'Hello World'
```



Length of string = 11

Grouping Data with Python Lists

```
In [1]: list_1 = [1, 2, 3]
        list_1

Out[1]: [1, 2, 3]
```

List of numbers

```
In [2]: list_2 = ['Hello', 'World']
        list_2

Out[2]: ['Hello', 'World']
```

List of strings

```
In [3]: list_3 = [1, 2, 3, 'Apple', 'orange']
        list_3

Out[3]: [1, 2, 3, 'Apple', 'orange']
```

List of numbers + strings

```
In [4]: list_4 = [list_1, list_2]
        list_4

Out[4]: [[1, 2, 3], ['Hello', 'World']]
```

List of lists

Indexing Lists

```
In [3]: list_3 = [1, 2, 3, 'Apple', 'orange']
        list_3

Out[3]: [1, 2, 3, 'Apple', 'orange']

In [5]: list_3[2]

Out[5]: 3

In [6]: list_3[:3]

Out[6]: [1, 2, 3]

In [7]: list_3[-1]

Out[7]: 'orange'

In [8]: list_3[-3:]

Out[8]: [3, 'Apple', 'orange']
```

1	2	3	‘Apple’	‘orange’
0	1	2	3	4

Index

More information on indexing:
https://railsware.com/blog/python-for-machine-learning-indexing-and-slicing-for-lists-tuples-strings-and-other-sequential-types/

Append, Insert, Delete List Elements

```
In [10]: list_3.append(4)
         list_3

Out[10]: [1, 2, 3, 'Apple', 'orange', 4]
```

Appending a new value

```
In [12]: list_3.insert(2,'pineapple')
         list_3

Out[12]: [1, 2, 'pineapple', 3, 'Apple', 'orange']
```

Inserting a new value into an index

2: Index to insert,

'pineapple': Value to insert

```
In [14]: del list_3[2]
         list_3

Out[14]: [1, 2, 'Apple', 'orange']
```

Deleting an existing value

2: Index to delete

Empty List and Element Check

```
In [15]: empty_list = []
         empty_list.append(5)
         empty_list

Out[15]: [5]
```

Appending a value to an empty list []

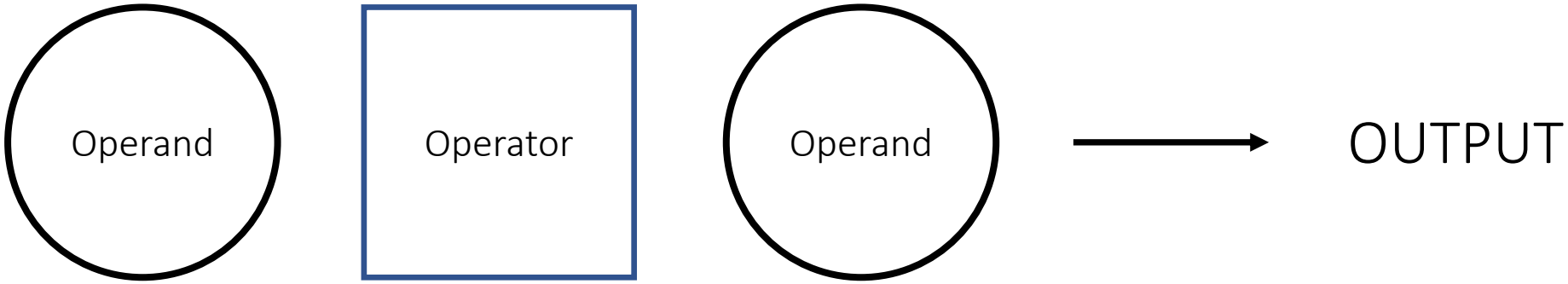
```
In [16]: 5 in empty_list

Out[16]: True
```

Checking if an element is in the list

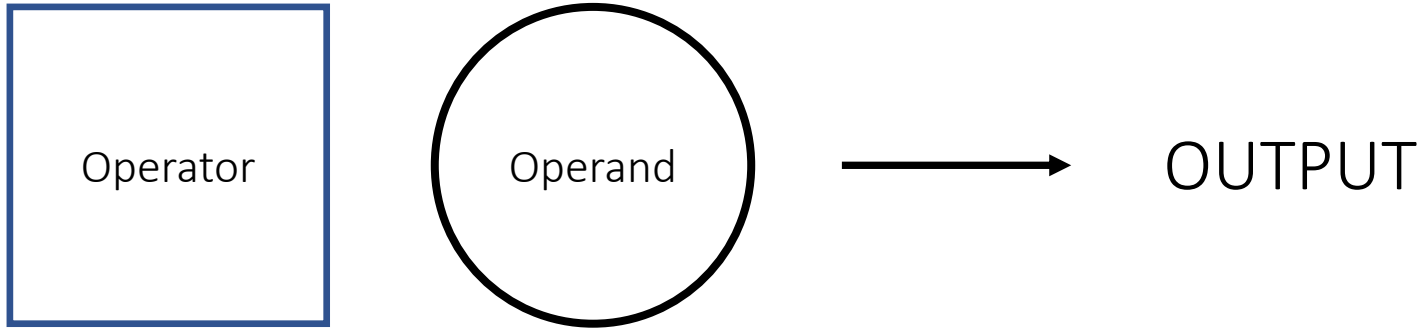
Operators in Python

Operators in Python

1. Binary Operator



2. Unary Operator



Arithmetic Operators

	Operator
Addition	+
Subtraction	-
Multiplication	*
Exponent	**
Division	/
Modulo	%

Example

```
float1, float2 = 5.4, 8.9
print(float1 + float2)
```

```
14.3
```

```
print(float1 - float2)
```

```
-3.5
```

```
print(float1 * float2)
```

```
48.06
```

```
print(float1**2)
```

```
29.160000000000004
```

```
print(float1 / float2)
```

```
0.6067415730337079
```

```
float1, float2 = 10., 3.
print(float1 % float2)
```

```
1.0
```

Comparison Operators

	Operator	Example
Greater Than	<	`5 < 3` → `False`
Less Than	>	`5 > 3` → `True`
Greater Than or Equal to	>=	`5 >= 3` → `True`
Less Than or Equal to	<=	`5 <= 3` → `False`
Equivalent to	==	`5 == 3` → `False`
Not Equivalent to	!=	`5 != 3` → `True`

Assignment Operators

	Operator	Example

Add and Assign	+=	`var1 = 3`
		`var1 += 1`
		`print(var1)`
		`4`

Subtract and Assign	-=	`var1 -= 1`
		`print(var1)`
		`3`

Multiply and Assign	*=	`var1 *= 1.5`
		`print(var1)`
		`4.5`

Divide and Assign	/=	`var1 /= 2`
		`print(var1)`
		`2.25`

Logical Operators

	Operator	Example
OR	or	
AND	and	
NOT	not	

```
bool1, bool2 = True, False
print(bool1 or bool2)
```

```
True
```

```
print(bool1 and bool2)
```

```
False
```

```
print(not bool1)
```

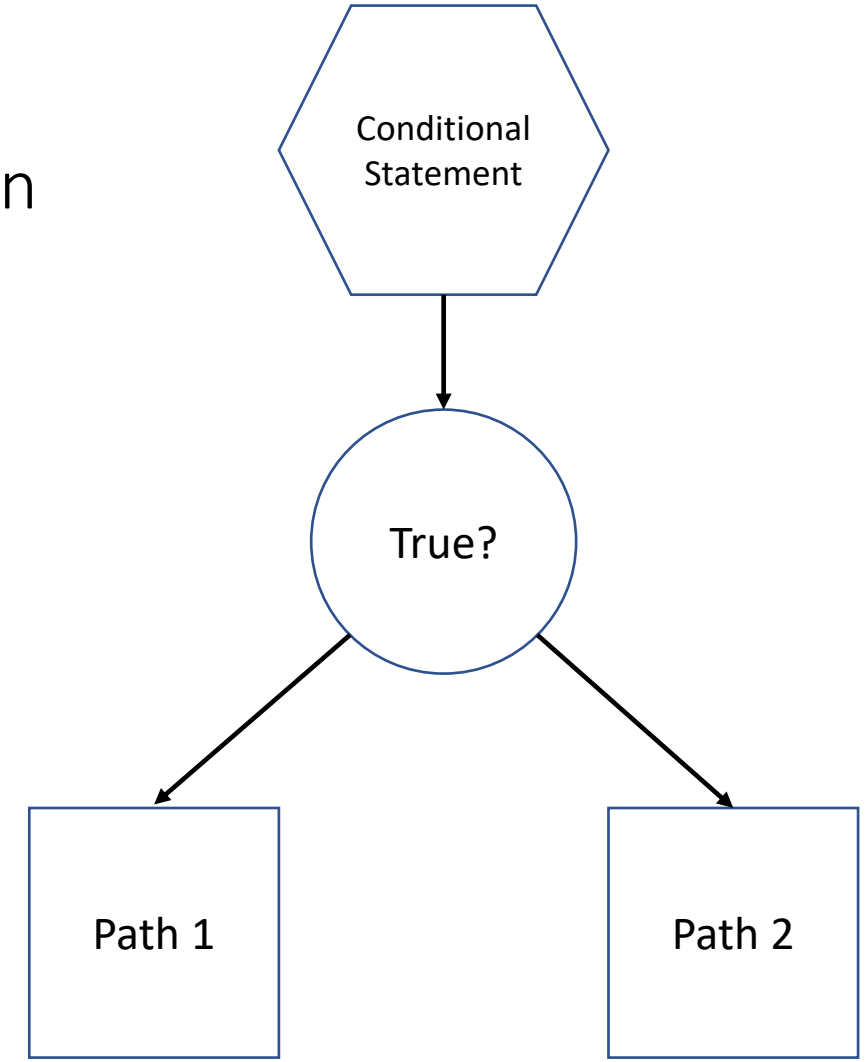
```
False
```

Conditionals, Loops, Functions

Conditional Statements

Types of conditional statements in Python

- If
- If-else
- If-elif-else



if statement

Implementation structure

If condition:

Code to be executed

Code example

In [11]:
```
num1 = 10
num2 = 20

if num1 < num2: # equivalently, if (num1 < num2) == True

    print('num2 is larger than num1')
```

```
num2 is larger than num1
```

In [2]:
```
if type(num1) == int:

    print('num1 is integer')
```

```
num1 is integer
```

If-else Statement

Implementation structure

If condition:

Execute this code

else:

Execute this code instead

Code example

```
In [5]: num1 = 20
        num2 = 10

        if num1 < num2:

            print('num2 is larger than num1')

        else:

            print('num2 is less or equal to num1')
```

```
num2 is less or equal to num1
```

If-elif-else Statement

Implementation structure

If condition 1:

Execute this code

elif condition 2:

Execute this code instead

else:

Execute this code instead

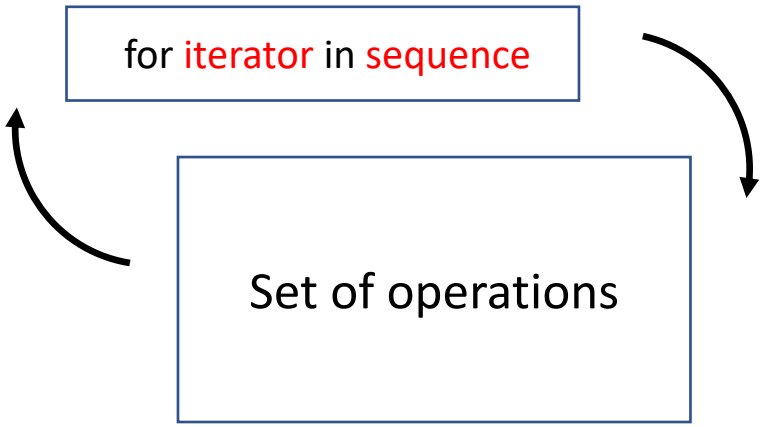
Code example

```
In [7]: num1 = 20

if type(num1) == float:

    print('num1 is float')

elif type(num1) == bool:

    print('num1 is boolean')

else:

    print('num1 is neither float nor boolean')
```

```
num1 is neither float nor boolean
```

Note: You can have multiple elif conditions between if and else

for Loop




```
for i in range(1, 11):  # A sequence from 1 to 10

    if i % 2 == 0:
        print(i, " is even")
    else:
        print(i, " is odd")
```

```
1  is odd
2  is even
3  is odd
4  is even
5  is odd
6  is even
7  is odd
8  is even
9  is odd
10  is even
```

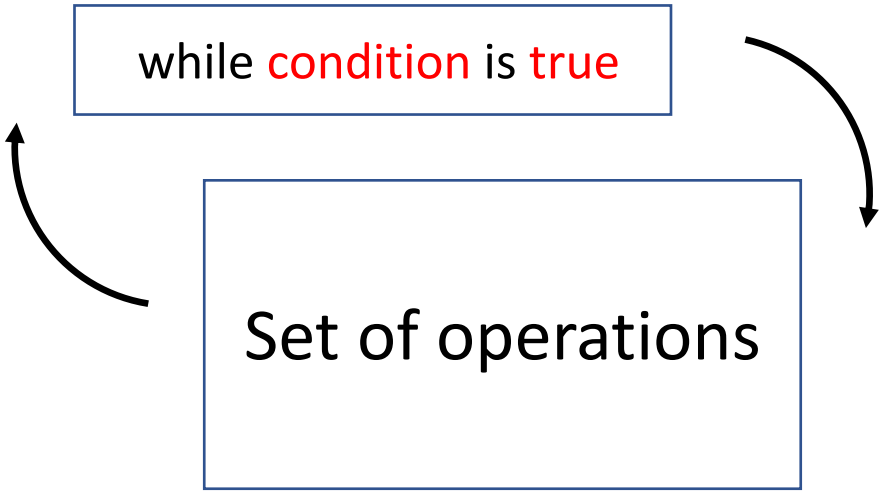
Iterate through sequence

```
# For loop - Iterate through list elements

float_list = [2.5, 16.42, 10.77, 8.3, 34.21]

for num in float_list:  # Iterator goes through each item in the list
    print([num, num * 2])
```

```
[2.5, 5.0]
[16.42, 32.84]
[10.77, 21.54]
[8.3, 16.6]
[34.21, 68.42]
```

Iterate through list elements

while Loop




```
In [43]: number_list = [1,2,3,4,5,6,7,8,9,10]

         k = 0

         while number_list[k] < 5:

             powered = number_list[k] ** 2

             print(powered)

             k += 1
```

```
1
4
9
16
```

```
In [1]: x = 1

        while(x > 0):

            print("This loop will never end!!")
```

```
This loop will never end!!
This loop will never end!!
This loop will never end!!
```

Note: while loop has a potential to run infinitely if not set correctly

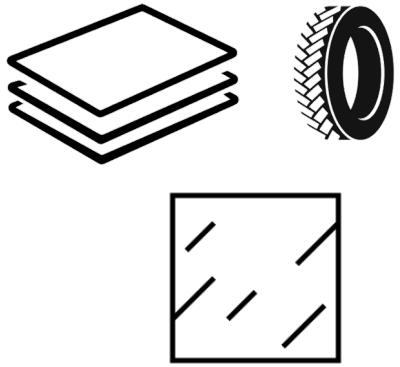
Functions

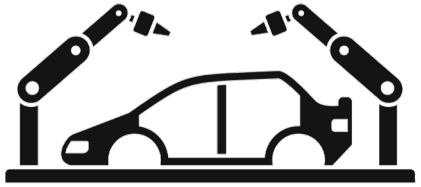
Function

INPUTS

Set of Instructions

OUTPUT





Defining Functions

Define function name Input parameters

In [16]:
```
def find_smaller_number(num1, num2):

    if num1 < num2:
        minimum = num1

    elif num1 == num2:
        minimum = 'two numbers are equal'

    else:
        minimum = num2

    return minimum
```

Set of instructions

Return output

Note: ‘return’ is NOT required for defining a function

PART 3:

NUMPY AND PLOTTING

Introduction to NumPy

Plotting with Matplotlib

Introduction to NumPy

What is NumPy?

Fundamental package for scientific computing in Python

- Supports multi-dimensional array object
- Provides assortment of mathematical routines for arrays
- Fast array operations through pre-compiled C
- Support array-wide broadcasting for operations
- Included in Anaconda 3

Constructing NumPy Arrays

From Python lists

```
import numpy as np

# 1D array
arr = np.array([1,2,3,4,5])

# 2D array
arr_2d = np.array([[1,2,3,4,5],
                   [6,7,8,9,10],
                   [11,12,13,14,15]])


print("Array dimensions: ", arr.shape)
print("Array dimensions: ", arr_2d.shape)
print("Array type: ", type(arr))
```

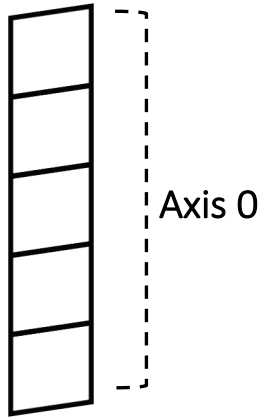
```
Array dimensions:  (5,)
Array dimensions:  (3, 5)
Array type:  <class 'numpy.ndarray'>
```

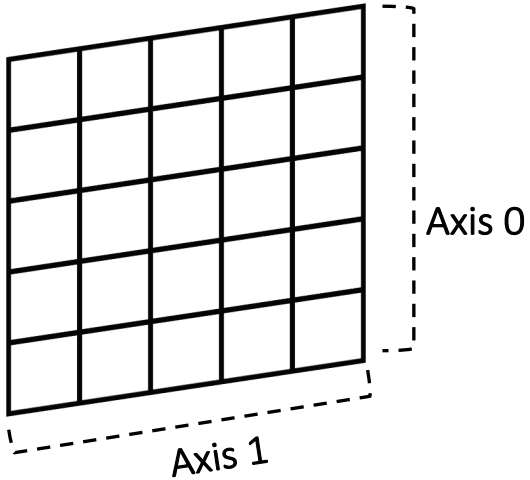
From Numpy commands

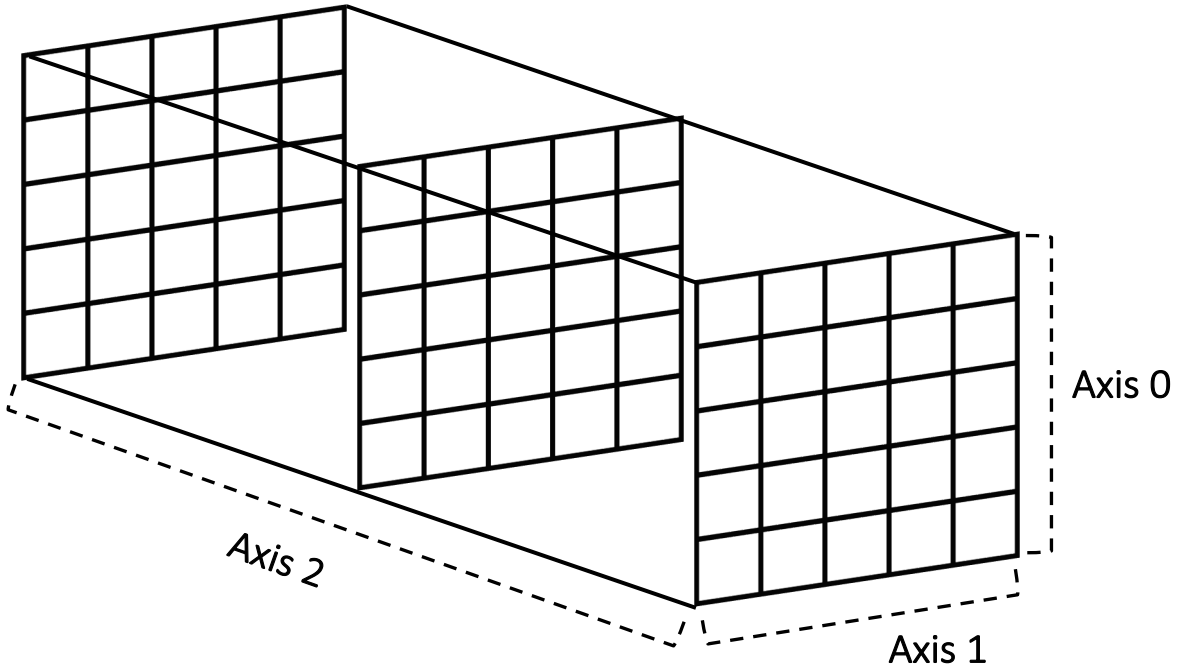
```
# Define number of each dimension
n1 = 3
n2 = 4

# Zeros array
zeros_1d = np.zeros(n1)
zeros_2d = np.zeros((n1,n2))

# Ones array
ones_1d = np.ones(n1)
ones_2d = np.ones((n1,n2))

# Creating array using np.arange
arr_arange = np.arange(0, 10, 1)     # (start, stop, stepsize)

# Creating an array using np.linspace
arr_linspace = np.linspace(0, 9, 10) # (start, stop, # of bins)

print("1D zeros array: ", zeros_1d)
print("1D ones array: ", ones_1d)
print("Number sequence from 0 to 9 using arange: ", arr_arange)
print("Number sequence from 0 to 9 using linspace: ", arr_linspace)
```

```
1D zeros array:  [0. 0. 0.]
1D ones array:  [1. 1. 1.]
Number sequence from 0 to 9 using arange:  [0 1 2 3 4 5 6 7 8 9]
Number sequence from 0 to 9 using linspace:  [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

Data Structures as Numpy Arrays







1-D

Shape = (i,)

e.g. time series data

2-D

Shape = (i,j)

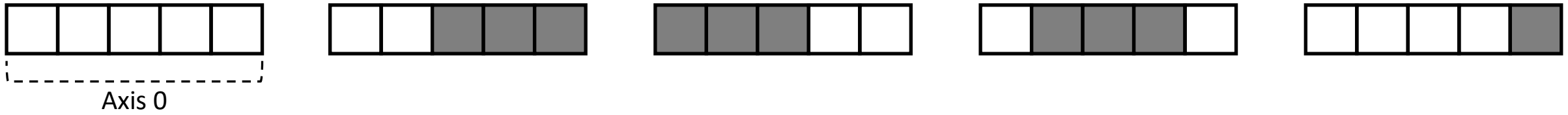
e.g. data frame, table, greyscale image

3-D

Shape = (i,j,k)

e.g. RGB color image

Slicing Arrays (1D)



arr

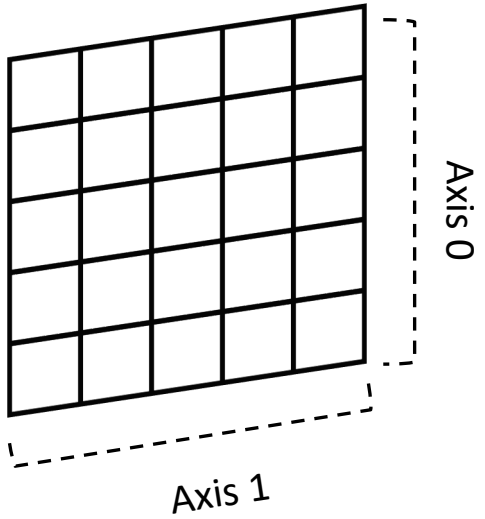
arr[2:]
=
arr[-3:]

arr[:3]
=
arr[:-2]

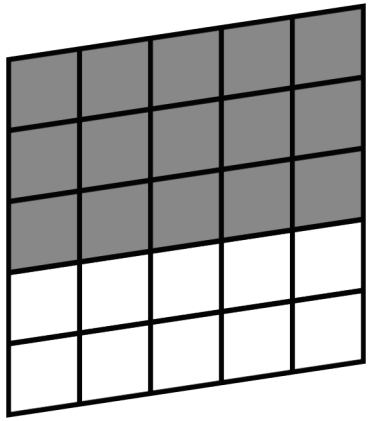
arr[1:4]
=
arr[-4:-1]

arr[4]
=
arr[-1]

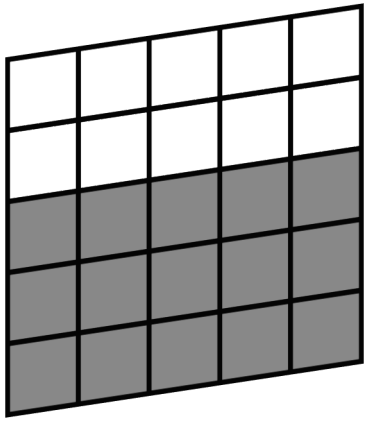
Slicing Arrays (2D)



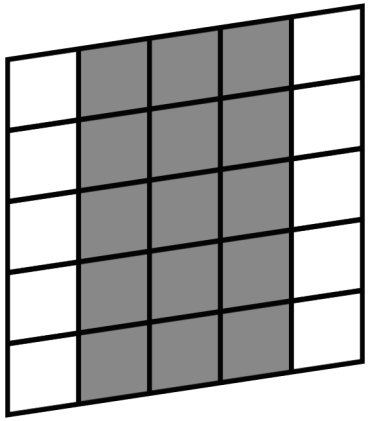
arr



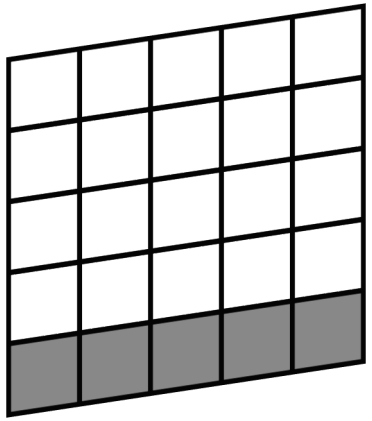
arr[:3]
||
arr[:3, :]
||
arr[:-2]
||
arr[:-2, :]



arr[2:]
||
arr[2:, :]
||
arr[-3:]
||
arr[-3:, :]

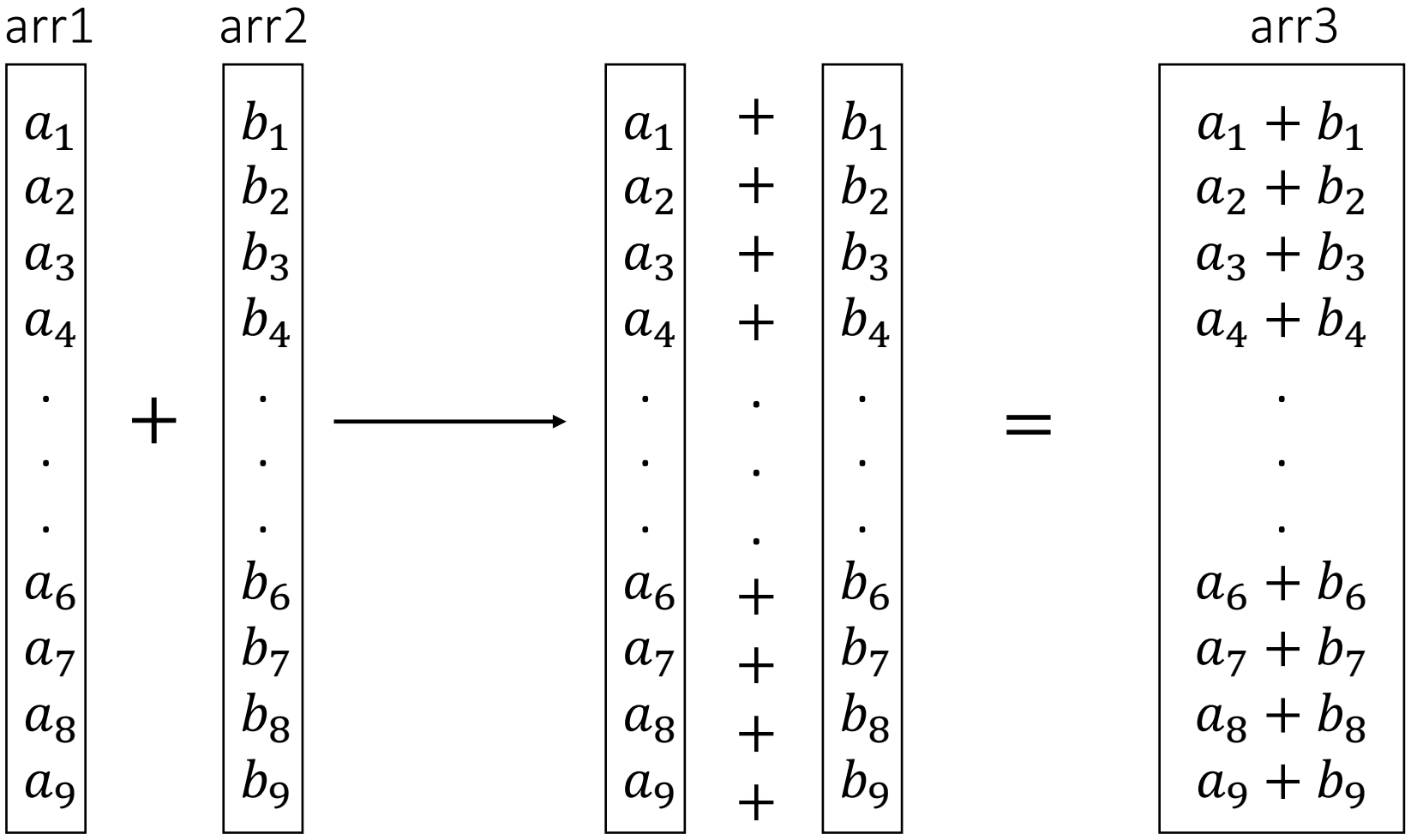


arr[:, 1:4]
||
arr[:, -4:-1]



arr[4]
||
arr[4, :]
||
arr[-1]
||
arr[-1, :]

Array-wide Operations in NumPy



numpy.add(arr1, arr2)

NumPy Arithmetic Operators

	Operator	Example
Addition	np.add()	

```
arr_1 = np.arange(0, 10, 1)  # 0 to 9
arr_2 = np.arange(10, 20, 1) # 10 to 19

print("arr_1 + arr_2:", np.add(arr_1, arr_2))
```

```
arr_1 + arr_2: [10 12 14 16 18 20 22 24 26 28]
```

Subtraction np.subtract()

```
print("arr_1 - arr_2:", np.subtract(arr_1, arr_2))
```

```
arr_1 - arr_2: [-10 -10 -10 -10 -10 -10 -10 -10 -10 -10]
```

Multiplication np.multiply()

```
print("arr_1 * arr_2:", np.multiply(arr_1, arr_2))
```

```
arr_1 * arr_2: [  0  11  24  39  56  75  96 119 144 171]
```

Note: The syntax assumes “import numpy as np”

NumPy Arithmetic Operators

	Operator	Example
Exponent	np.exp()	

```
print("exp(arr_1):", np.exp(arr_1)[:5]) # Print first 5
```

```
exp(arr_1): [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
```

	Operator	Example
Division	np.divide()	

```
print("arr_1 / arr_2:", np.divide(arr_1, arr_2)[:5]) # Print first 5
```

```
arr_1 / arr_2: [0.         0.09090909 0.16666667 0.23076923 0.28571429]
```

	Operator	Example
Modulo	np.mod()	

```
print("10 % 3:", np.mod(10, 3))
```

```
10 % 3: 1
```

Math Operators

	Operator	Example
Sine	np.sin(x)	
Cosine	np.cos(x)	
Tangent	np.tan(x)	

```
x_arr = np.array([1,2,3])

print(np.sin(x_arr))
```

```
[0.84147098 0.90929743 0.14112001]
```

```
print(np.cos(x_arr))
```

```
[ 0.54030231 -0.41614684 -0.9899925 ]
```

```
print(np.tan(x_arr))
```

```
[ 1.55740772 -2.18503986 -0.14254654]
```

Note: Trigonometric functions require radians as inputs

Math Operators

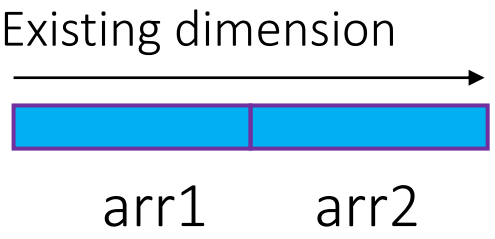
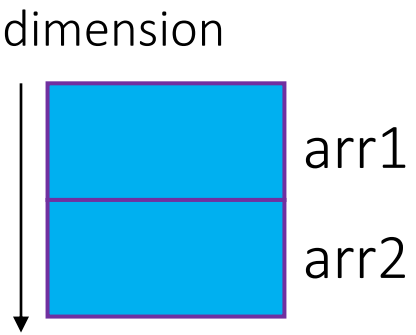
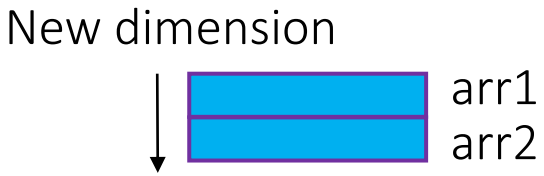
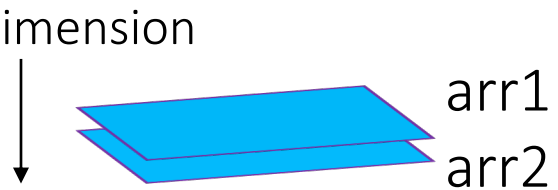
	Operator	Example
Pi	np.pi	`print(np.pi)` → `3.141592653589793`
Square Root	np.sqrt(x)	`print(np.sqrt(x_arr))` → `[1. 1.41421356 1.73205081]`

More functions: https://numpy.org/doc/stable/reference/routines.math.html

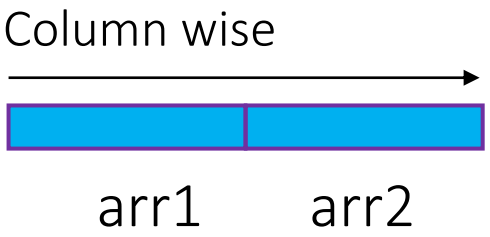
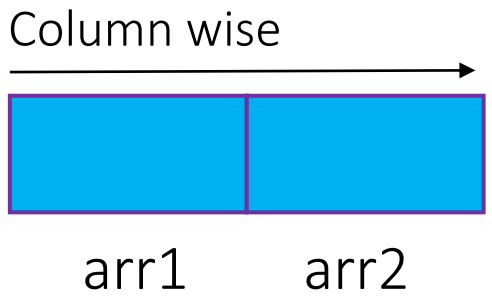
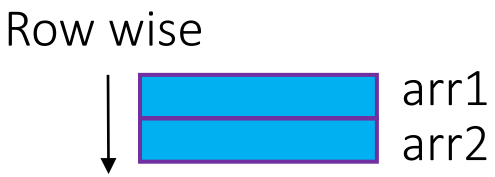
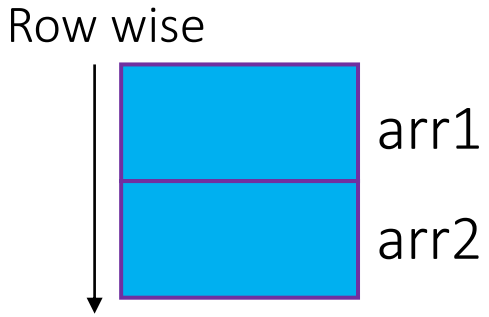
Combining Arrays

	Operator	Example
Concatenation	np.concatenate()	`print(np.concatenate([arr_1, arr_2]))` → `[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]`
Stack Dimensions	np.stack()	`print(np.stack([arr_1, arr_2]))` → `[[ 0 1 2 3 4 5 6 7 8 9]` ` [10 11 12 13 14 15 16 17 18 19]]`
Horizontal Stack	np.hstack()	`print(np.hstack([arr_1, arr_2]))` → `[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]`
Vertical Stack	np.vstack()	`print(np.vstack([arr_1, arr_2]))` → `[[ 0 1 2 3 4 5 6 7 8 9]` ` [10 11 12 13 14 15 16 17 18 19]]`

Combining Arrays

Operator	1D	2D
np.concatenate()		
np.stack()		

Combining Arrays

Operator	1D	2D
np.hstack()		
np.vstack()		

Array Splitting

	Operator	Example
Split the array into sub-arrays (axis defines direction)	np.split()	
Split the array column-wise	np.hsplit()	
Split the array row-wise	np.vsplit()	

```
1 print(array_2d)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
1 np.split(array_2d, 2, axis = 0)
```

```
[array([[0, 1, 2, 3],
        [4, 5, 6, 7]]),
 array([[ 8,  9, 10, 11],
        [12, 13, 14, 15]])]
```

```
1 print(np.hsplit(array_2d, 2))
```

```
[array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]]), array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])]
```

```
1 print(np.vsplit(array_2d, 2))
```

```
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
```

Characteristic Values of Arrays

	Operator	Example
Minimum Value	np.min()	`print(np.min(arr_1))` → `0`
Maximum Value	np.max()	`print(np.max(arr_1))` → `9`
Mean Value	np.mean()	`print(np.mean(arr_1))` → `4.5`
Summed Value	np.sum()	`print(np.sum(arr_1))` → `45`

Note: axis parameter allows you to compute characteristic value alongside specific axis - e.g. np.sum(arr_1, axis =0): summation along row axis.

Indexing Arrays

	Operator	Example

Minimum Value Index	np.argmin()	`arr_3 = np.array([4,2,6,7,8,9,3])`
		`print(np.argmin(arr_3))`
		`1`

Maximum Value Index	np.argmax()	`print(np.argmax(arr_3))` `5`
Sort Indices (low to high)	np.argsort()	`print(np.argsort(arr_3))` `[1 6 0 2 3 4 5]`
Find Indices satisfying a Condition	np.where()	`print(np.where(arr_3 < 7))` `(array([0, 1, 2, 6], dtype=int64),)`

Plotting with Matplotlib

Basic Plotting with Matplotlib

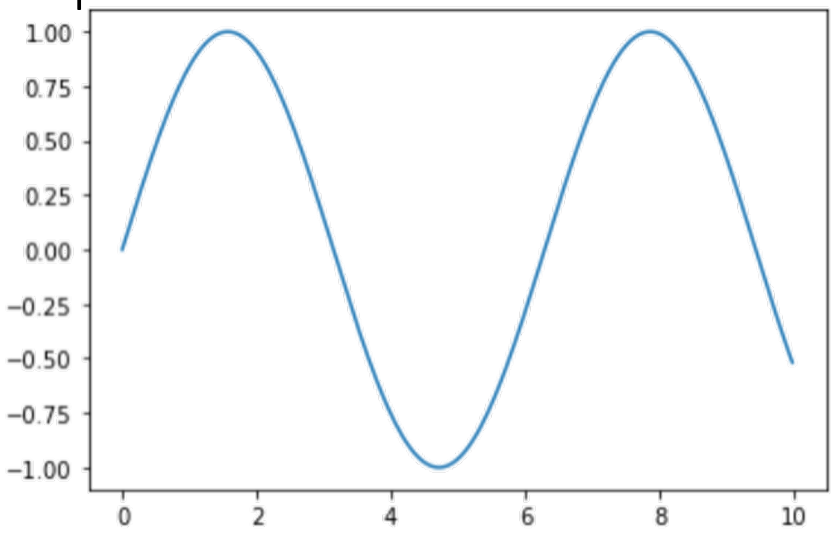
Import Matplotlib

```
#%matplotlib inline   # If using local notebook runtime, allows you to display the plot inside the jupyter notebook
#%matplotlib notebook # Alternatively, you can use this line instead for interactive plots

import matplotlib.pyplot as plt
```

Code

```
x = np.arange(0, 10, 1/32) # x axis data
y = np.sin(x)              # y axis data
plt.plot(x, y)             # plot the data
```

Output

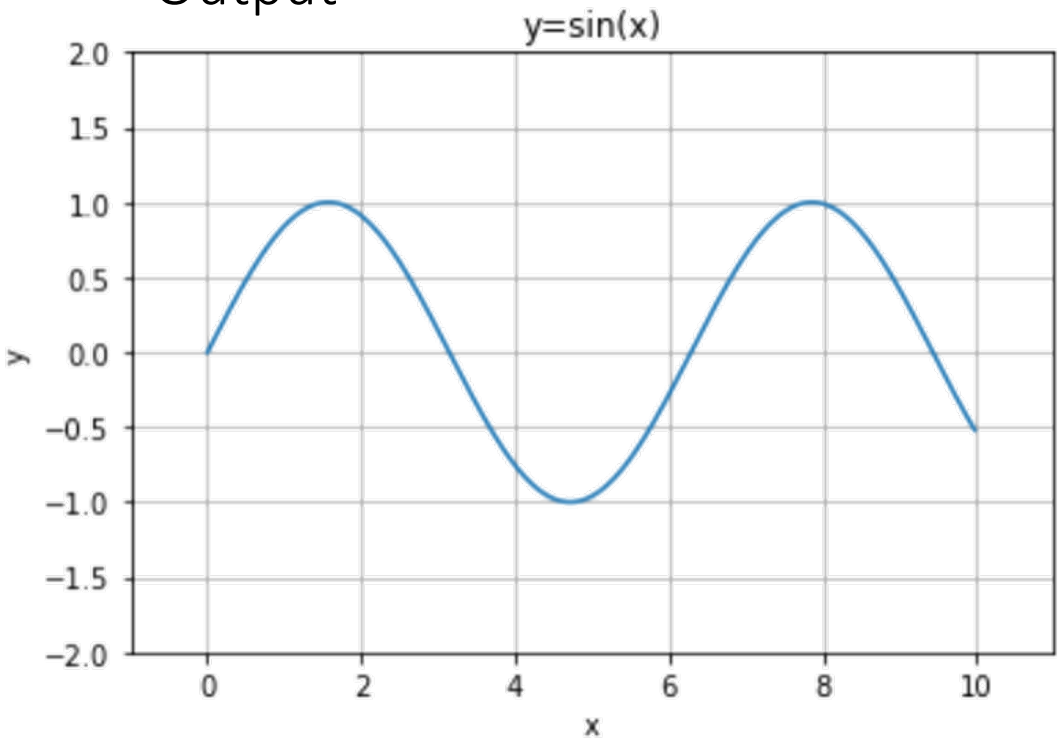


Labeling Your Plots

Code

```
plt.plot(x, y)
plt.title('y=sin(x)')  # set the title
plt.xlabel('x')        # set the x axis label
plt.ylabel('y')        # set the y axis label
plt.xlim(-1, 11)       # set the x axis range
plt.ylim(-2, 2)        # set the y axis range
plt.grid()             # enable the grid
```

Output

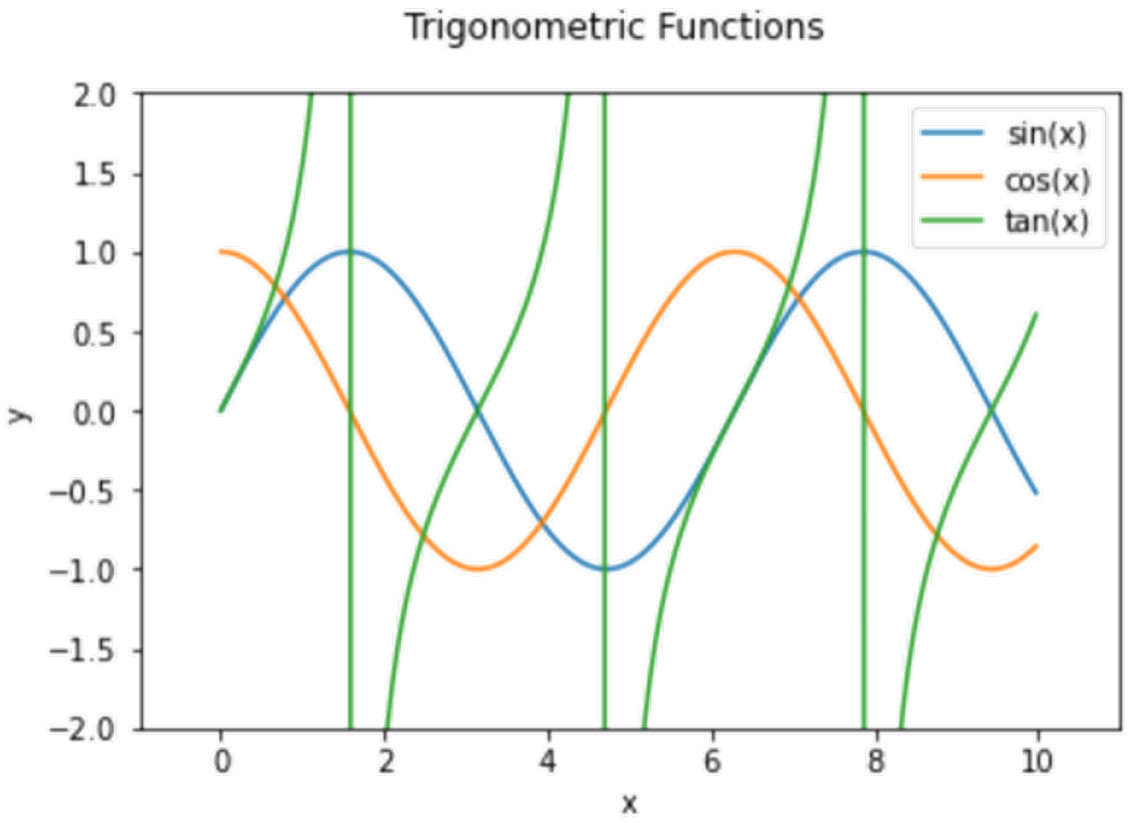


Multiple Plots

Code

```
# Multiple Plots
# On same figure
x = np.arange(0, 10, 1/32) # x axis data
y1 = np.sin(x)             # y axis data 1
y2 = np.cos(x)             # y axis data 2
y3 = np.tan(x)             # y axis data 3
plt.figure(1)              # create figure 1
plt.plot(x, y1, label='sin(x)')
plt.plot(x, y2, label='cos(x)')
plt.plot(x, y3, label='tan(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-1, 11)
plt.ylim(-2, 2)
plt.suptitle('Trigonometric Functions')
plt.legend()
plt.show()
```

Output



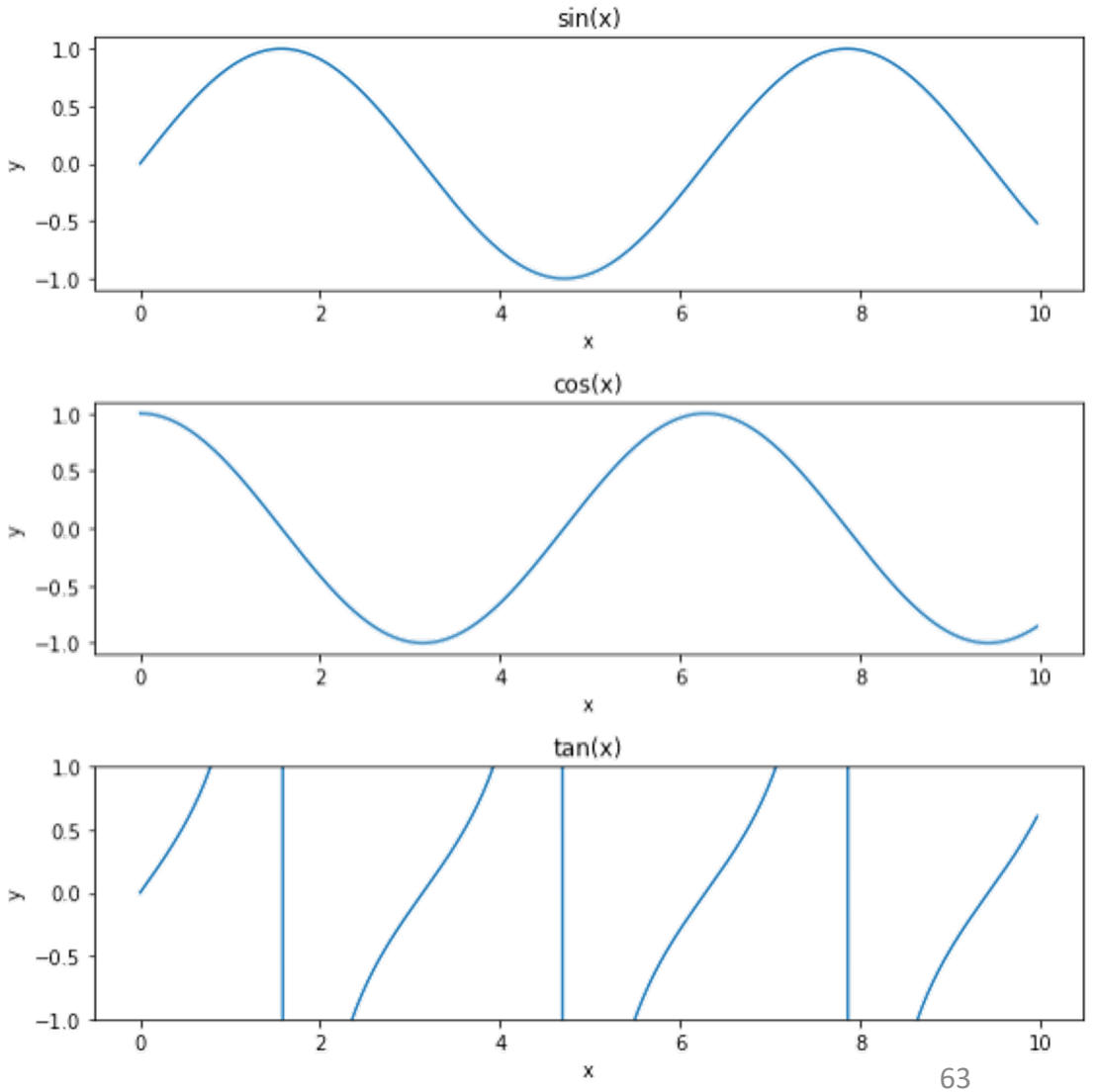
Creating Subplots

Code

```
# Multiple Subplots
x = np.arange(0, 10, 1/32) # x axis data
y1 = np.sin(x)             # y axis data for subplot 1
y2 = np.cos(x)             # y axis data for subplot 2
y3 = np.tan(x)             # y axis data for subplot 3

fig = plt.figure(2,figsize=(8,8)) # create figure 2

plt.subplot(311)           # (number of rows, number of columns, current plot)
plt.plot(x, y1)
plt.title('sin(x)')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(312)
plt.plot(x, y2)
plt.title('cos(x)')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(313)
plt.plot(x, y3)
plt.title('tan(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(-1, 1)

fig.tight_layout()
```

Official documentation:
https://matplotlib.org/stable/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py

Output



LAB 1 SUPPLEMENTARY:

BASIC DEBUGGING WITH PYTHON

General Tips on Minimizing Errors

Debugging with ‘print’

Debugging with PDB

Using Google/Stack Overflow

General Tips on Minimizing Errors

Do not panic when you get errors

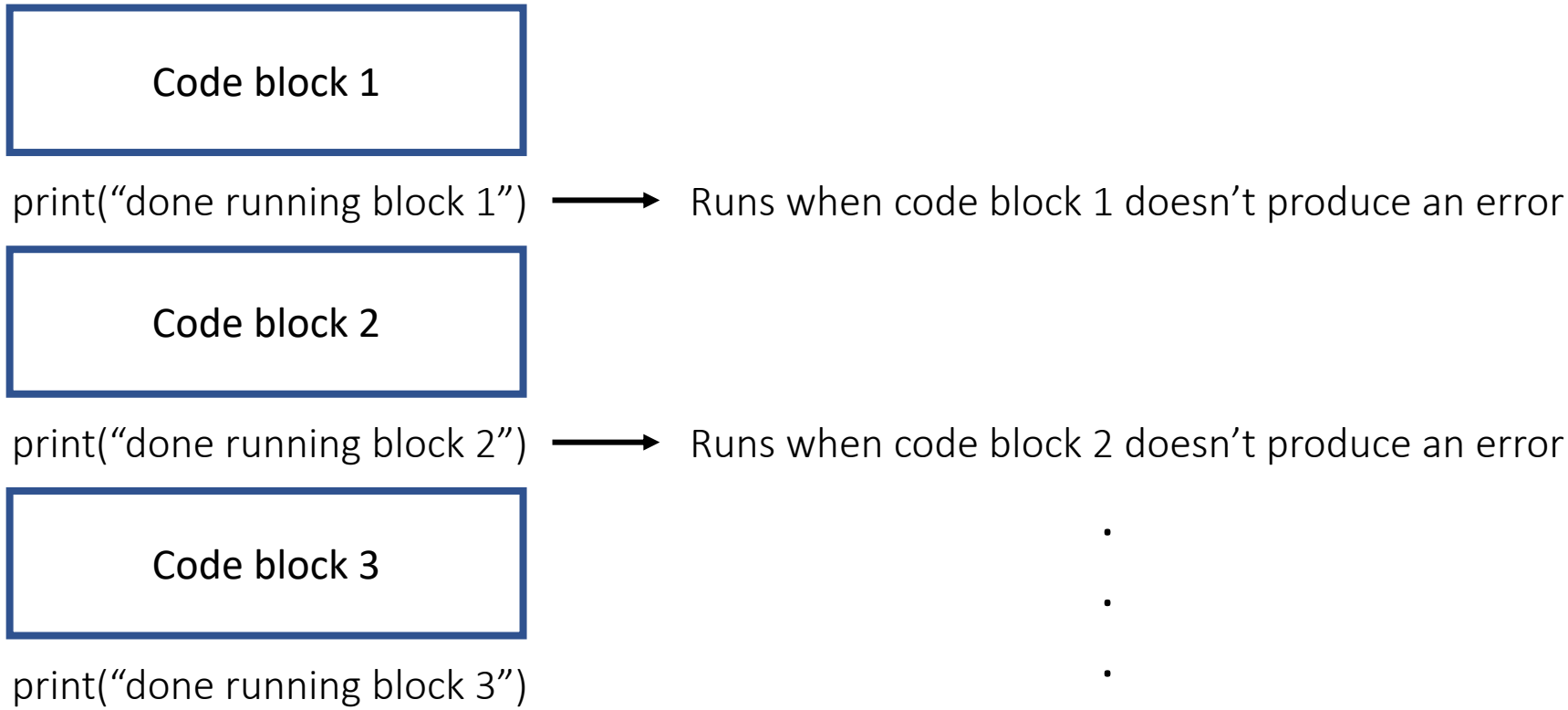
Outline your code structure ahead of time

Keep your code organized

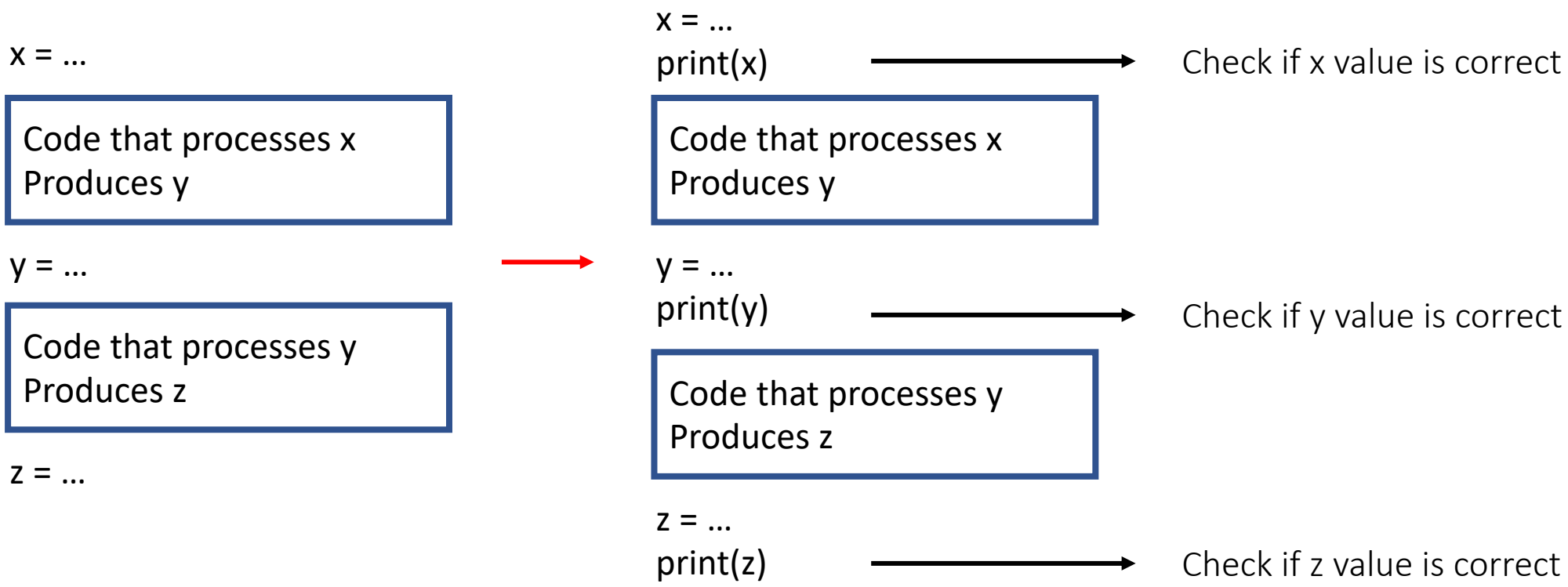
Test your code often

More tips on avoiding errors by Berkeley online textbook
https://pythonnumericalmethods.berkeley.edu/notebooks/chapter10.00-Errors-Practices-Debugging.html

Basic Debugging with ‘print’



Basic Debugging with ‘print’



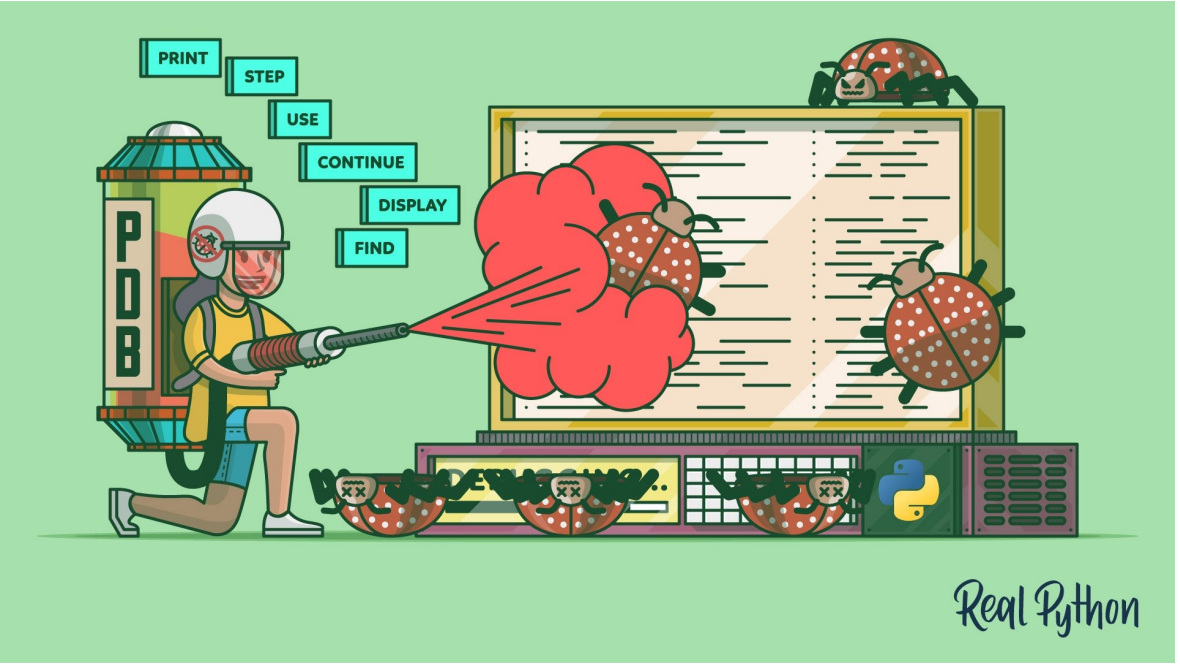
Original code

Debugging each step with print()

Python Debugger

What is Python Debugger (PDB)?

- Debugging tool built in Python
- No installation required
- Provides Interactive environment
- Widely used for systemic debugging



PDB Basics

```
def divide(divisor):

    breakpoint()

    val = 1/divisor

    return val
```

Upon execution of the code, activates Python debugger console at this line

```
divide(0)
```

```
> <ipython-input-24-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     return val
```

```
ipdb>
```

Python debugger console

PDB Useful Commands

```
> <ipython-input-24-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     return val

ipdb>
```

- h: help
- w: where
- n: next
- c: continue
- p: print
- l: list
- q: quit

PDB Useful Commands

```
divide(0)
```

```
> <ipython-input-24-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     return val

ipdb> h

Documented commands (type help <topic>):
========================================
EOF     commands   enable    ll        pp       s                 until
a       condition  exit      longlist  psource  skip_hidden       up
alias   cont       h         n         q        skip_predicates   w
args    context    help      next      quit     source            whatis
b       continue   ignore    p         r        step              where
break   d          interact  pdef      restart  tbreak
bt      debug      j         pdoc      return   u
c       disable    jump      pfile     retval   unalias
cl      display    l         pinfo     run      undisplay
clear   down       list      pinfo2    rv       unt

Miscellaneous help topics:
==========================
exec  pdb

ipdb> h l
Print lines of code from the current stack frame

ipdb>
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Provide documentation for the command

PDB Useful Commands

```
divide(2)

> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb> w
    [... skipping 27 hidden frame(s)]
  <ipython-input-2-cfdb0f794c84>(1)<module>()
----> 1 divide(2)

> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb>
```

h: help

w: where

n: next

c: continue

p: print

l: list

q: quit

Print a stack trace (i.e. the order of code being executed), with the most recent frame at the bottom.

PDB Useful Commands

```
divide(2)

> <ipython-input-1-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     return val

ipdb> n
> <ipython-input-1-03e5b3dd8265>(7)divide()
      3     breakpoint()
      4
      5     val = 1/divisor
      6
----> 7     return val

ipdb>
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Executes the next line of code

PDB Useful Commands

```
divide(2)

> <ipython-input-1-a669feb31165>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     breakpoint()

ipdb> c
> <ipython-input-1-a669feb31165>(9)divide()
      5     val = 1/divisor
      6
      7     breakpoint()
      8
----> 9     return val

ipdb>
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Run the code until the next breakpoint()

PDB Useful Commands

```
divide(2)
```

```
> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb> p val
0.5

ipdb>
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Print the value of the desired variable

PDB Useful Commands

```
divide(2)

> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb> l
      2
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb>
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Similar to where but print the lines of code from the current stack frame

PDB Useful Commands

```
divide(2)
```

```
> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb> q

---------------------------------------------------------------------------
BdbQuit                                   Traceback (most recent call last)
<ipython-input-2-cfdb0f794c84> in <module>
----> 1 divide(2)

<ipython-input-1-f4a9c3842530> in divide(divisor)
      5     breakpoint()
      6
----> 7     return val

<ipython-input-1-f4a9c3842530> in divide(divisor)
      5     breakpoint()
      6
----> 7     return val

~\anaconda3\lib\bdb.py in trace_dispatch(self, frame, event, arg)
     86             return # None
     87         if event == 'line':
---> 88             return self.dispatch_line(frame)
     89         if event == 'call':
     90             return self.dispatch_call(frame, arg)

~\anaconda3\lib\bdb.py in dispatch_line(self, frame)
    111         if self.stop_here(frame) or self.break_here(frame):
    112             self.user_line(frame)
--> 113             if self.quitting: raise BdbQuit
    114         return self.trace_dispatch
    115

BdbQuit:
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Exit the debugger console

Useful video tutorial of basic usage of Python debugger:
https://www.youtube.com/watch?v=aZJnGOwzHtU

Using Google/Stack Overflow