Business Research and Data Analytics

Lecture 10: Python's Functions and Files. NumPy Basics.

Iegor Vyshnevskyi

Woosong University

May 14/18, 2024

Agenda

1. Python Functions
2. Files and the Operating System
3. NumPy Intro
4. Your first NumPy array
5. NumPy data types
6. Selecting and Updating Data
7. Array Mathematics
8. Array Transformations
9. In-class Assignment

1. Python Functions

Functions basics

- Functions are the primary and most important method of code organization and reuse in Python.
- As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function.
- Functions can also help make your code more readable by giving a name to a group of Python statements.

Source: Wes McKinney (2022)

Functions basics

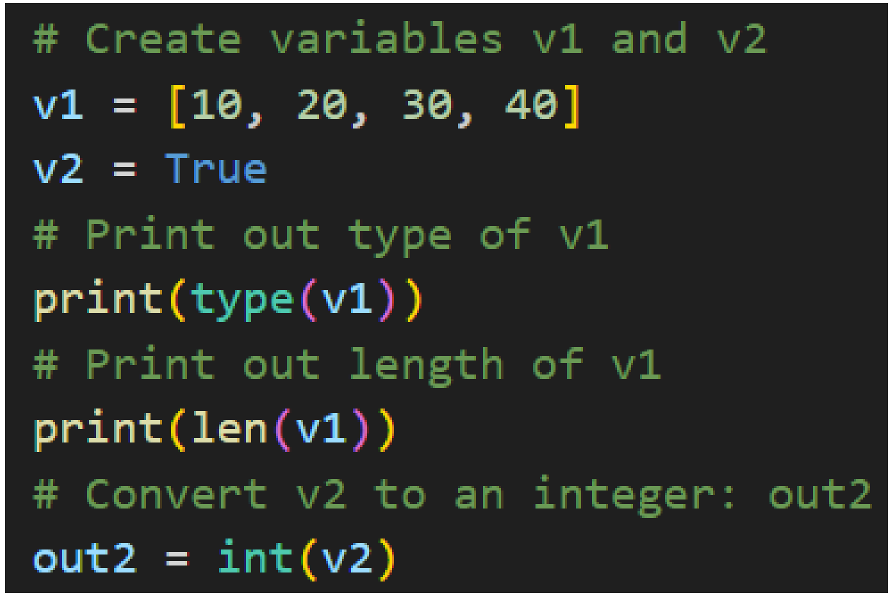
In-built functions

- Python comes equipped with a variety of built-in functions designed to simplify the tasks of a data scientist.
- You're already familiar with a couple of these functions: print() and type().
- Additionally, you've employed functions like str(), int(), bool(), and float() to facilitate the conversion between different data types. These functions are also part of Python's built-in arsenal.
- The general way for calling functions and saving the result to a variable is: *output = function_name(input)*

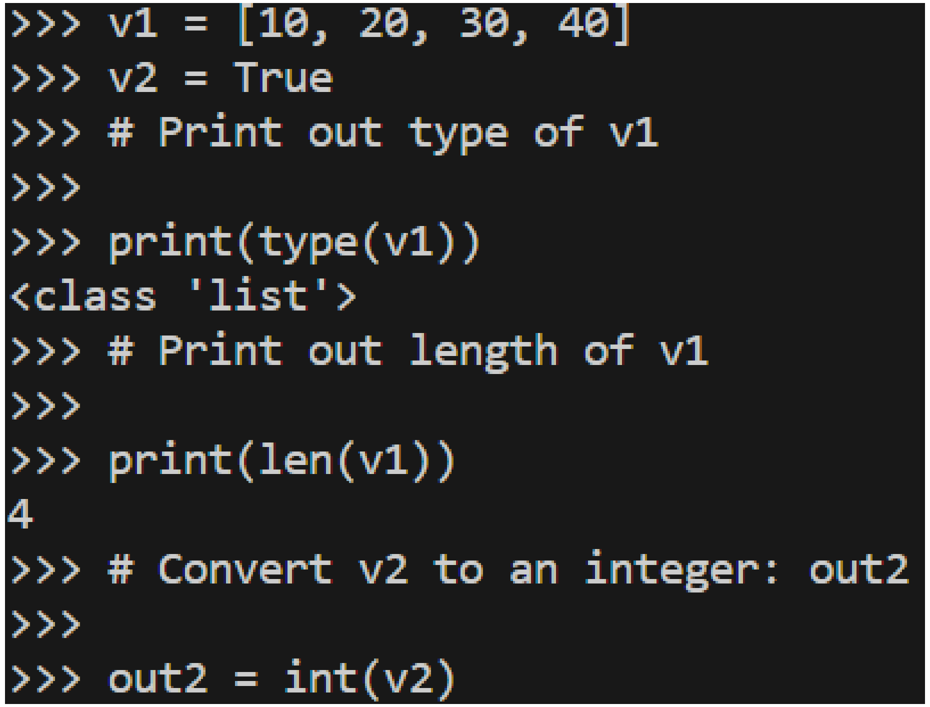
Source: Wes McKinney (2022)

Functions basics

Example



```python
# Create variables v1 and v2
v1 = [10, 20, 30, 40]
v2 = True
# Print out type of v1
print(type(v1))
# Print out length of v1
print(len(v1))
# Convert v2 to an integer: out2
out2 = int(v2)
```



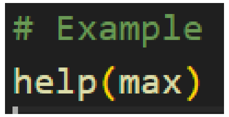
```
>>> v1 = [10, 20, 30, 40]
>>> v2 = True
>>> # Print out type of v1
>>>
>>> print(type(v1))
<class 'list'>
>>> # Print out length of v1
>>>
>>> print(len(v1))
4
>>> # Convert v2 to an integer: out2
>>>
>>> out2 = int(v2)
```

Source: DataCamp; Wes McKinney (2022)

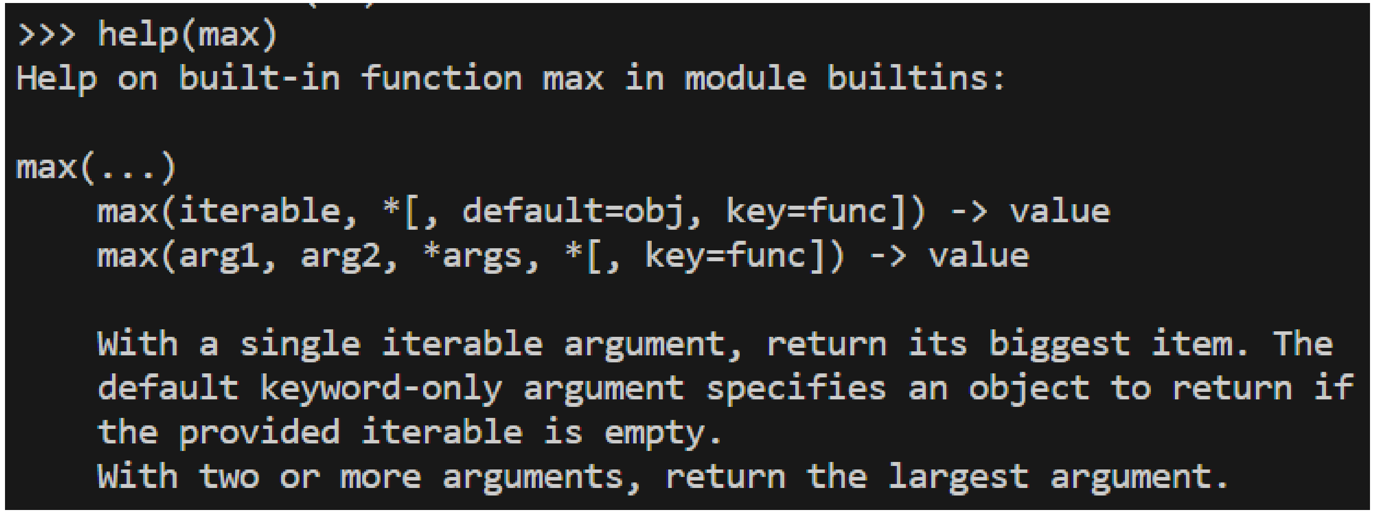
Functions basics

Help function

- To find details about any function you may use help function:



```python
# Example
help(max)
```



```
>>> help(max)
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Multiple arguments

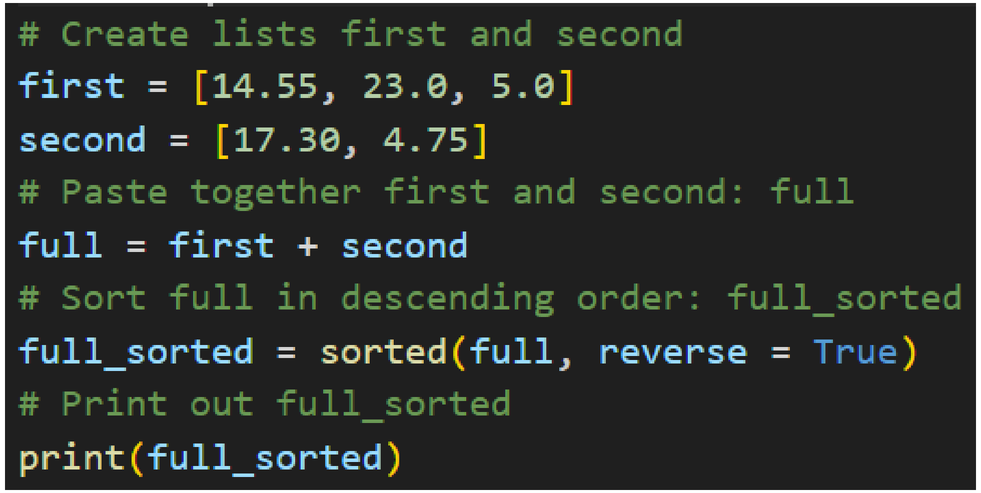
- A function may have different attributes/arguments.
- Take a moment to review the documentation for the sorted() function by entering help(sorted) in your Python environment.
- You will observe that sorted() accepts three parameters: iterable, key, and reverse.
- key=None means that if you don't specify the key argument, it will be None.
- reverse=False means that if you don't specify the reverse argument, it will be False, by default.

Source: DataCamp; Wes McKinney (2022)

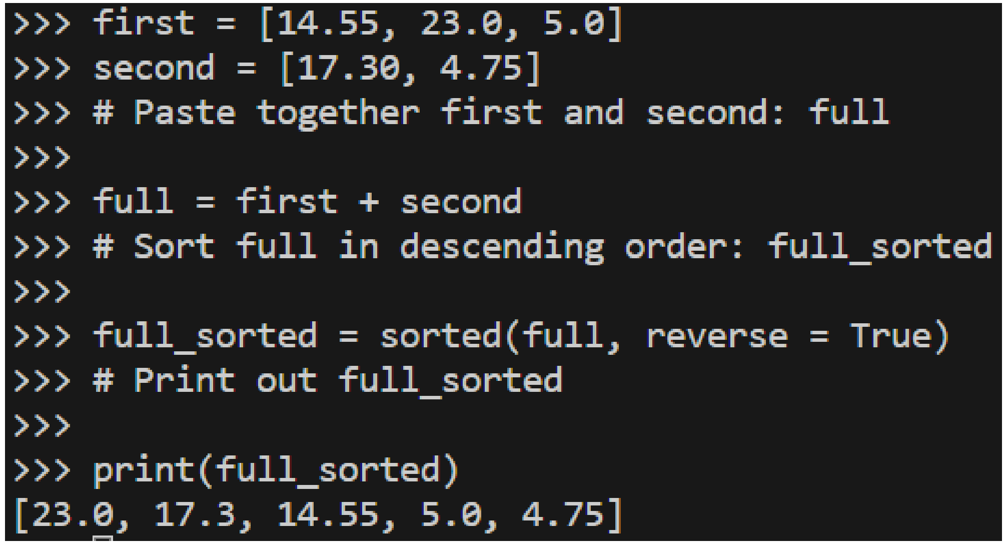
Functions basics

Multiple arguments

- Example:



```
# Create lists first and second
first = [14.55, 23.0, 5.0]
second = [17.30, 4.75]
# Paste together first and second: full
full = first + second
# Sort full in descending order: full_sorted
full_sorted = sorted(full, reverse = True)
# Print out full_sorted
print(full_sorted)
```



```
>>> first = [14.55, 23.0, 5.0]
>>> second = [17.30, 4.75]
>>> # Paste together first and second: full
>>>
>>> full = first + second
>>> # Sort full in descending order: full_sorted
>>>
>>> full_sorted = sorted(full, reverse = True)
>>> # Print out full_sorted
>>>
>>> print(full_sorted)
[23.0, 17.3, 14.55, 5.0, 4.75]
```

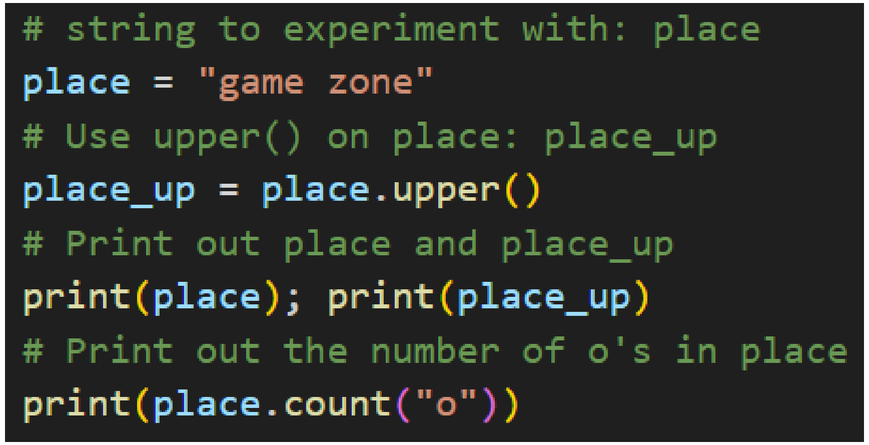
Source: DataCamp; Wes McKinney (2022)

Functions basics

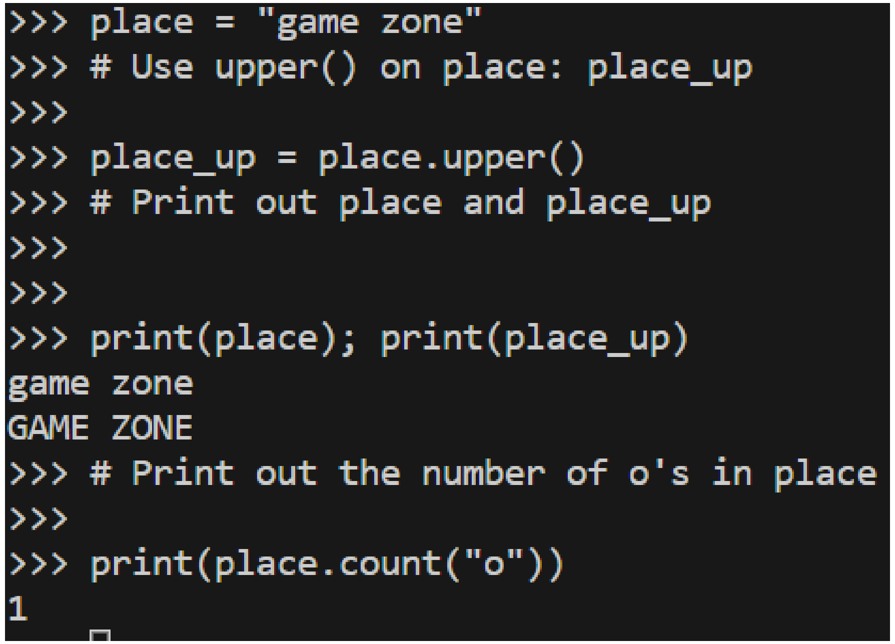
String Methods

- Strings are equipped with a variety of methods.

Example



```
# string to experiment with: place
place = "game zone"
# Use upper() on place: place_up
place_up = place.upper()
# Print out place and place_up
print(place); print(place_up)
# Print out the number of o's in place
print(place.count("o"))
```



```
>>> place = "game zone"
>>> # Use upper() on place: place_up
>>>
>>> place_up = place.upper()
>>> # Print out place and place_up
>>>
>>>
>>> print(place); print(place_up)
game zone
GAME ZONE
>>> # Print out the number of o's in place
>>>
>>> print(place.count("o"))
1
```

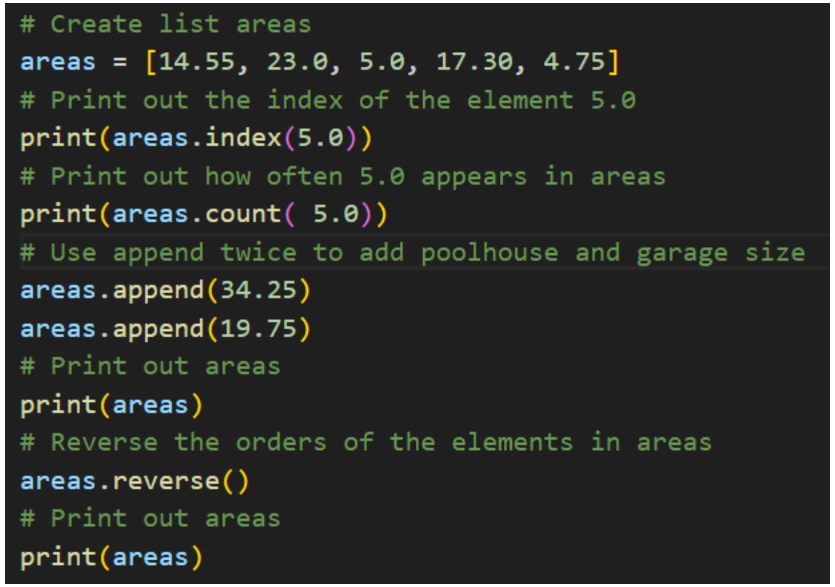
Source: DataCamp; Wes McKinney (2022)

Functions basics

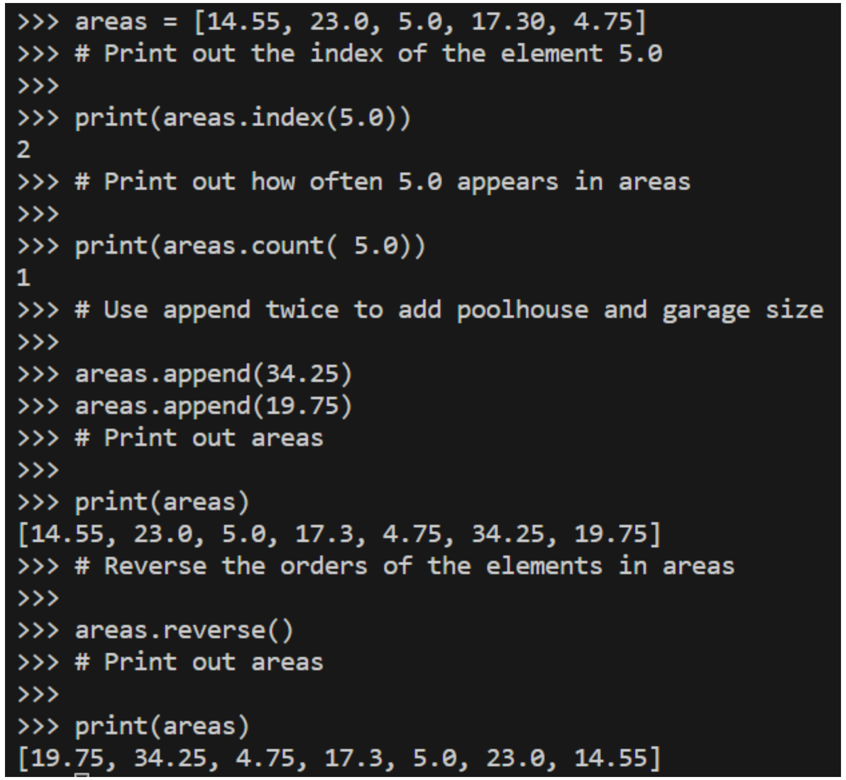
List Methods

- Strings aren't the sole Python types that feature associated methods. Lists, floats, integers, and booleans are also data types that include a plethora of helpful methods. In this exercise, you'll have the opportunity to explore:

Example



```python
# Create list areas
areas = [14.55, 23.0, 5.0, 17.30, 4.75]
# Print out the index of the element 5.0
print(areas.index(5.0))
# Print out how often 5.0 appears in areas
print(areas.count( 5.0))
# Use append twice to add poolhouse and garage size
areas.append(34.25)
areas.append(19.75)
# Print out areas
print(areas)
# Reverse the orders of the elements in areas
areas.reverse()
# Print out areas
print(areas)
```



```
>>> areas = [14.55, 23.0, 5.0, 17.30, 4.75]
>>> # Print out the index of the element 5.0
>>>
>>> print(areas.index(5.0))
2
>>> # Print out how often 5.0 appears in areas
>>>
>>> print(areas.count( 5.0))
1
>>> # Use append twice to add poolhouse and garage size
>>>
>>> areas.append(34.25)
>>> areas.append(19.75)
>>> # Print out areas
>>>
>>> print(areas)
[14.55, 23.0, 5.0, 17.3, 4.75, 34.25, 19.75]
>>> # Reverse the orders of the elements in areas
>>>
>>> areas.reverse()
>>> # Print out areas
>>>
>>> print(areas)
[19.75, 34.25, 4.75, 17.3, 5.0, 23.0, 14.55]
```

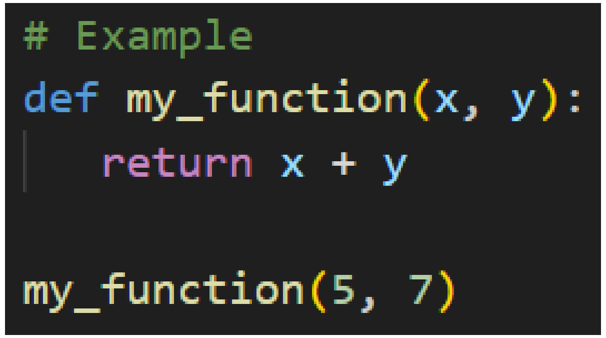
Source: DataCamp; Wes McKinney (2022)

Functions basics

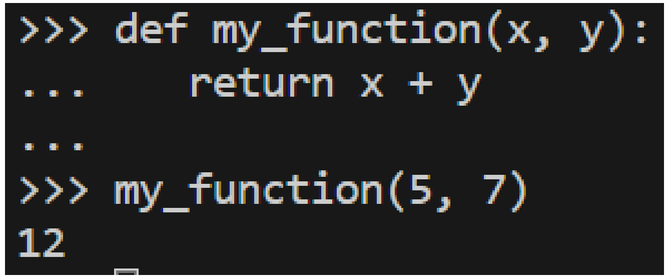
Writing a function

- Sometimes it’s more efficient to write own function.

Example



```python
# Example
def my_function(x, y):
    return x + y

my_function(5, 7)
```



```
>>> def my_function(x, y):
...     return x + y
...
>>> my_function(5, 7)
12
```

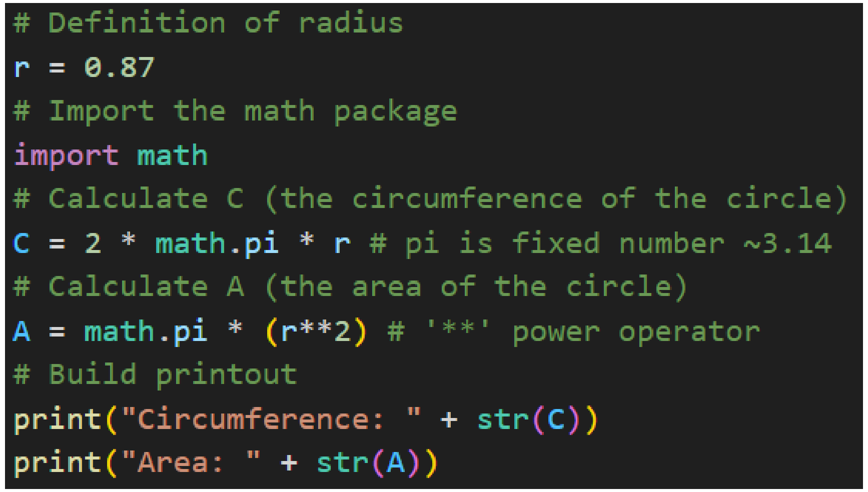
Source: DataCamp; Wes McKinney (2022)

Functions basics

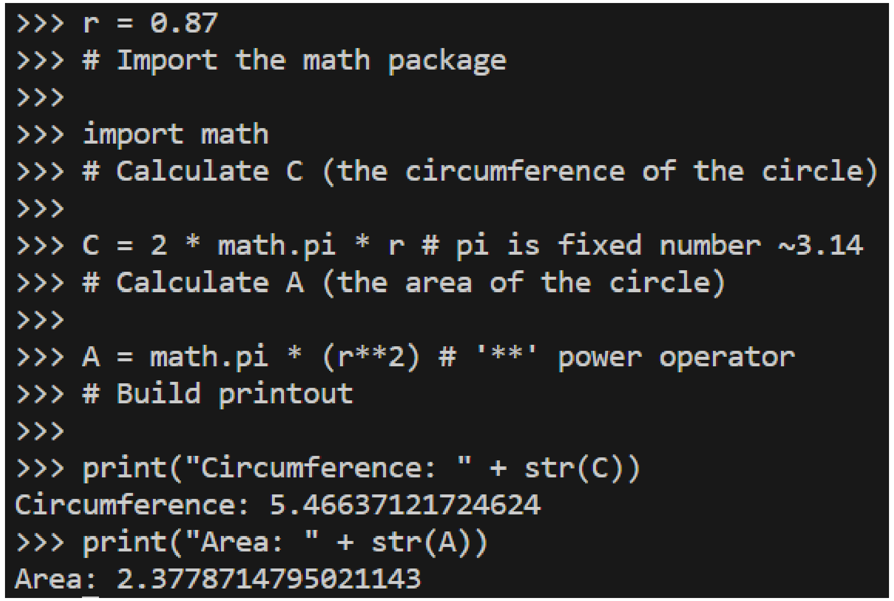
Import package

- You can get much more functions from some specific packages.

Example



```
# Definition of radius
r = 0.87
# Import the math package
import math
# Calculate C (the circumference of the circle)
C = 2 * math.pi * r # pi is fixed number ~3.14
# Calculate A (the area of the circle)
A = math.pi * (r**2) # '**' power operator
# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```



```
>>> r = 0.87
>>> # Import the math package
>>>
>>> import math
>>> # Calculate C (the circumference of the circle)
>>>
>>> C = 2 * math.pi * r # pi is fixed number ~3.14
>>> # Calculate A (the area of the circle)
>>>
>>> A = math.pi * (r**2) # '**' power operator
>>> # Build printout
>>>
>>> print("Circumference: " + str(C))
Circumference: 5.46637121724624
>>> print("Area: " + str(A))
Area: 2.3778714795021143
```

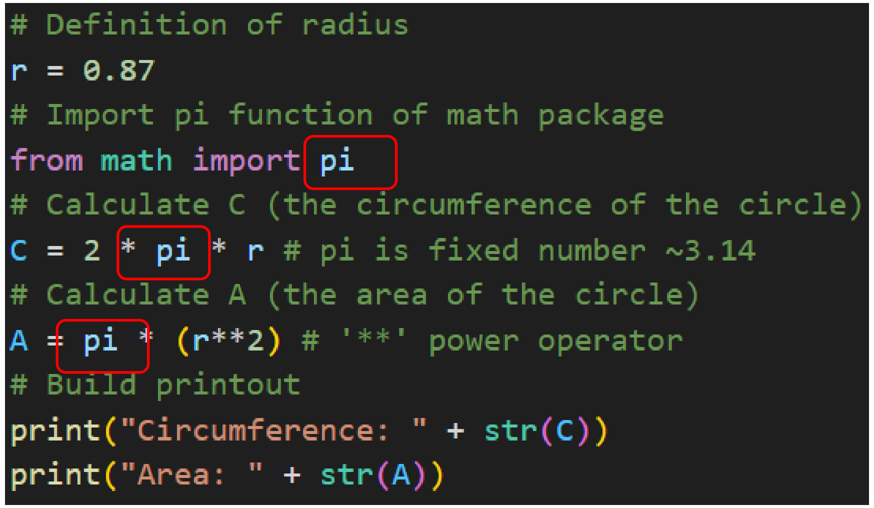
Source: DataCamp; Wes McKinney (2022)

Functions basics

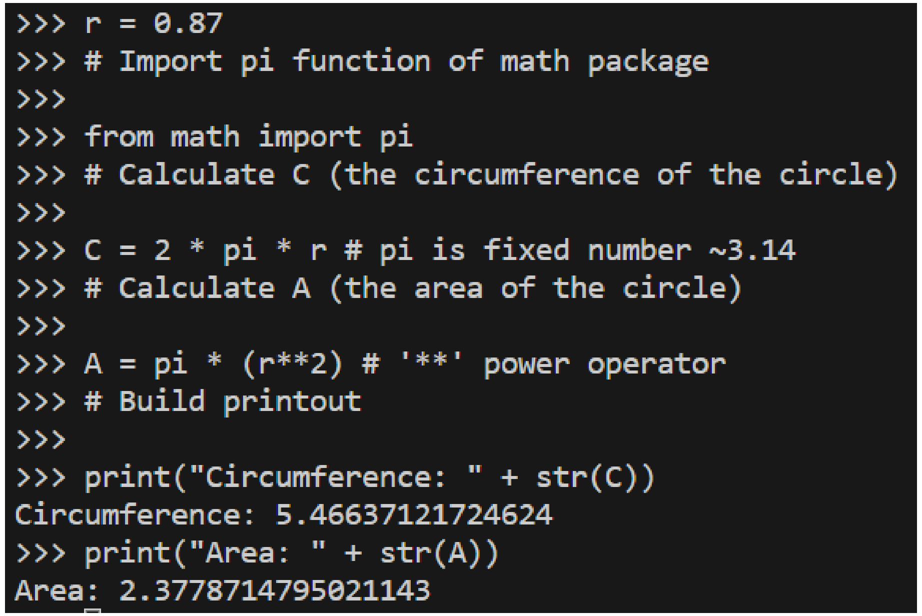
Selective import from package

- When you make general imports, such as import math, you gain access to all the functionality provided by the entire math package. However, if you prefer to use only a particular component of a package, you have the option to make your import more specific: ‘from math import pi’

Example



```
# Definition of radius
r = 0.87
# Import pi function of math package
from math import pi
# Calculate C (the circumference of the circle)
C = 2 * pi * r # pi is fixed number ~3.14
# Calculate A (the area of the circle)
A = pi * (r**2) # '**' power operator
# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```



```
>>> r = 0.87
>>> # Import pi function of math package
>>>
>>> from math import pi
>>> # Calculate C (the circumference of the circle)
>>>
>>> C = 2 * pi * r # pi is fixed number ~3.14
>>> # Calculate A (the area of the circle)
>>>
>>> A = pi * (r**2) # '**' power operator
>>> # Build printout
>>>
>>> print("Circumference: " + str(C))
Circumference: 5.46637121724624
>>> print("Area: " + str(A))
Area: 2.3778714795021143
```

Source: DataCamp; Wes McKinney (2022)

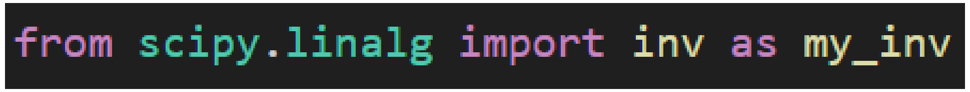
Functions basics

Different ways of importing

- There exist multiple approaches to importing packages and modules into Python. Depending on the specific import statement you choose, you will need to employ different Python code.

Example

- Let's say you want to use the inv() function, which is located in the *linalg* subpackage of the *scipy* package, then to get it you run:



```
from scipy.linalg import inv as my_inv
```

Source: DataCamp; Wes McKinney (2022)

2. Files and the Operating System

Files basics

It’s important to understand the basics of how to work with files in Python.

Fortunately, it’s relatively *straightforward*, which is one reason Python is so popular for text and file munging.

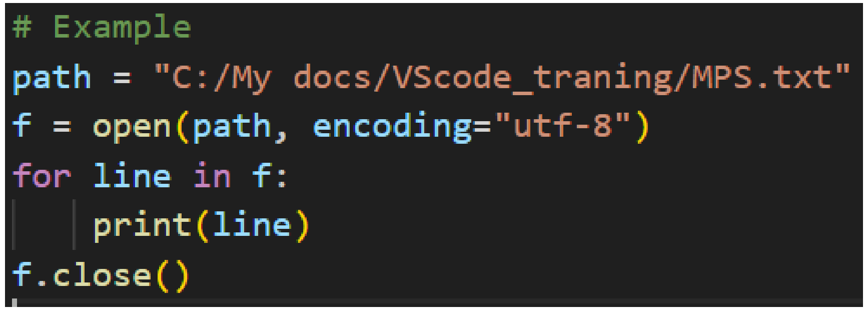
- To open a file for reading or writing, use the built-in open function with either a relative or absolute file path and an optional file encoding.
- We pass encoding="utf-8" as a best practice because the default Unicode encoding for reading files varies from platform to platform.
- By default, the file is opened in read-only mode "r".

Source: Wes McKinney (2022)

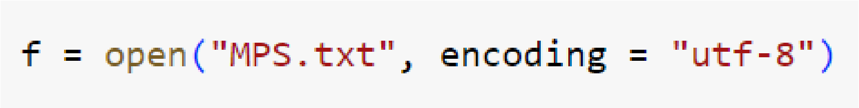
Files basics

Example

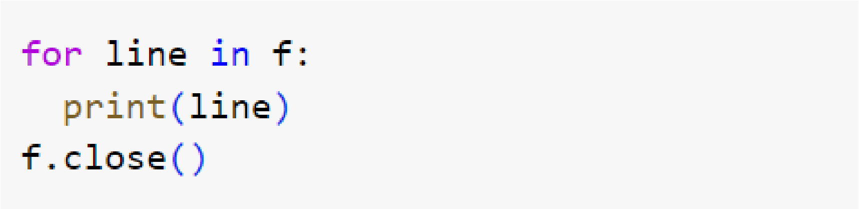
- We can then treat the file object f like a list and iterate over the lines like so:



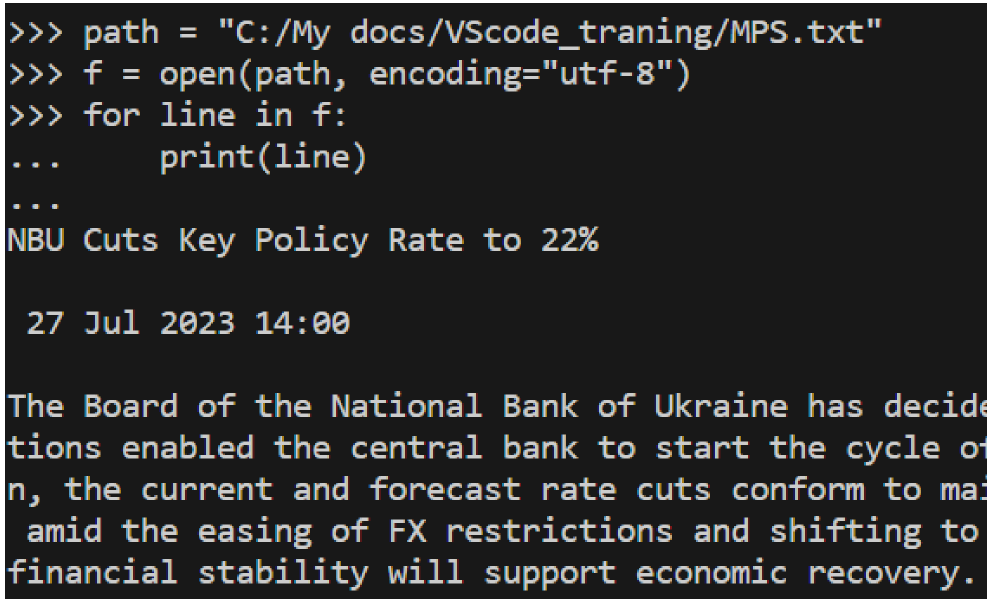
```python
# Example
path = "C:/My docs/VScode_traning/MPS.txt"
f = open(path, encoding="utf-8")
for line in f:
    print(line)
f.close()
```



```python
f = open("MPS.txt", encoding = "utf-8")
```



```python
for line in f:
  print(line)
f.close()
```

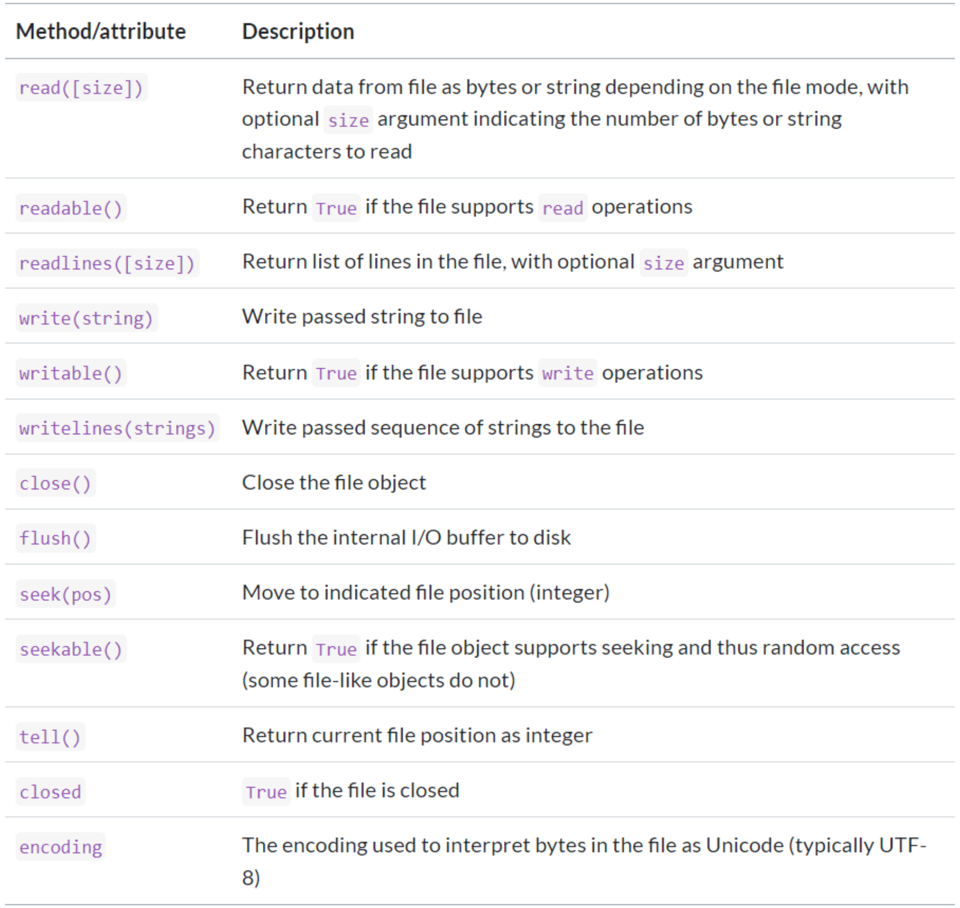


```
>>> path = "C:/My docs/VScode_traning/MPS.txt"
>>> f = open(path, encoding="utf-8")
>>> for line in f:
...     print(line)
...
NBU Cuts Key Policy Rate to 22%

 27 Jul 2023 14:00

The Board of the National Bank of Ukraine has decide
tions enabled the central bank to start the cycle of
n, the current and forecast rate cuts conform to mai
 amid the easing of FX restrictions and shifting to
financial stability will support economic recovery.
```

Source: Wes McKinney (2022)

Files basics

Important Python file methods or attributes



Method/attribute	Description
`read([size])`	Return data from file as bytes or string depending on the file mode, with optional `size` argument indicating the number of bytes or string characters to read
`readable()`	Return `True` if the file supports `read` operations
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`write(string)`	Write passed string to file
`writable()`	Return `True` if the file supports `write` operations
`writelines(strings)`	Write passed sequence of strings to the file
`close()`	Close the file object
`flush()`	Flush the internal I/O buffer to disk
`seek(pos)`	Move to indicated file position (integer)
`seekable()`	Return `True` if the file object supports seeking and thus random access (some file-like objects do not)
`tell()`	Return current file position as integer
`closed`	`True` if the file is closed
`encoding`	The encoding used to interpret bytes in the file as Unicode (typically UTF-8)

Source: Wes McKinney (2022)

3. NumPy Intro

Intro

NumPy, short for ***Numerical Python***, is one of the most important foundational packages for numerical computing in Python.

Many computational packages providing scientific functionality use NumPy's array objects as one of the standard interface for data exchange.

TensorFlow and scikit-learn use NumPy arrays as inputs, and pandas and Matplotlib are built on top of NumPy.

Source: Wes McKinney (2022); Introduction to NumPy. Izzy Weber. Rpubs

Intro

Here are some of the things you'll find in NumPy:

- ndarray, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible broadcasting capabilities
- Mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading/writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- a C API for connecting NumPy with libraries written in C, C++, or FORTRAN

Source: Wes McKinney (2022)

Intro

While NumPy by itself does not provide modeling or scientific functionality, having an *understanding of NumPy arrays and array-oriented computing* will *help you use tools with array computing semantics, like pandas*, much more effectively.

One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data.

Source: Wes McKinney (2022)

Application

For most data analysis applications, the main NumPy areas of functionality are:

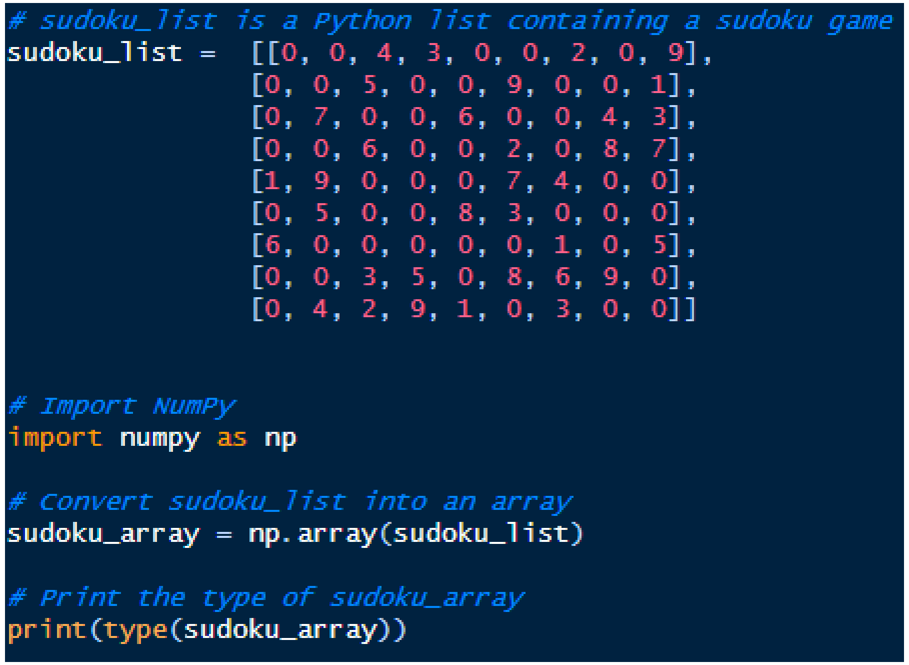
- Fast array-based operations for data munging and cleaning, subsetting and filtering, transformation, and any other kind of computation
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining heterogeneous datasets
- Expressing conditional logic as array expressions instead of loops with '*if-elif-else*' branches
- Group-wise data manipulations (aggregation, transformation, and function application)

Source: Wes McKinney (2022)

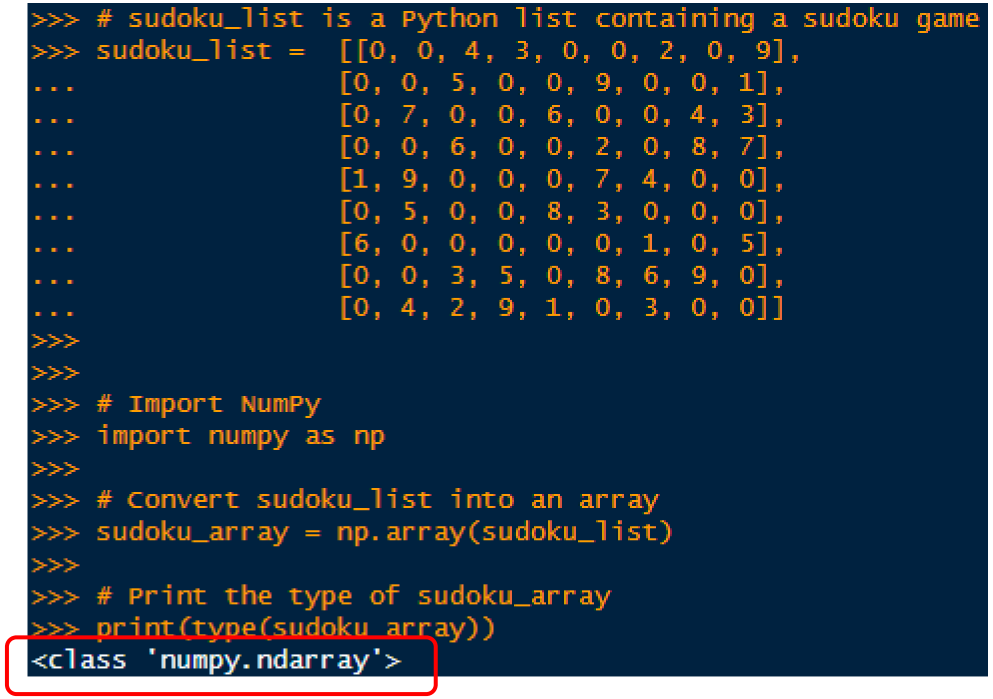
4. Your first NumPy array

Our First NumPy Array

We create NumPy array (*sudoku_array*) from the list. In this case *sudoku_list* is a Python list containing a sudoku game:



```
# sudoku_list is a Python list containing a sudoku game
sudoku_list =   [[0, 0, 4, 3, 0, 0, 2, 0, 9],
                [0, 0, 5, 0, 0, 9, 0, 0, 1],
                [0, 7, 0, 0, 6, 0, 0, 4, 3],
                [0, 0, 6, 0, 0, 2, 0, 8, 7],
                [1, 9, 0, 0, 0, 7, 4, 0, 0],
                [0, 5, 0, 0, 8, 3, 0, 0, 0],
                [6, 0, 0, 0, 0, 0, 1, 0, 5],
                [0, 0, 3, 5, 0, 8, 6, 9, 0],
                [0, 4, 2, 9, 1, 0, 3, 0, 0]]


# Import NumPy
import numpy as np

# Convert sudoku_list into an array
sudoku_array = np.array(sudoku_list)

# Print the type of sudoku_array
print(type(sudoku_array))
```

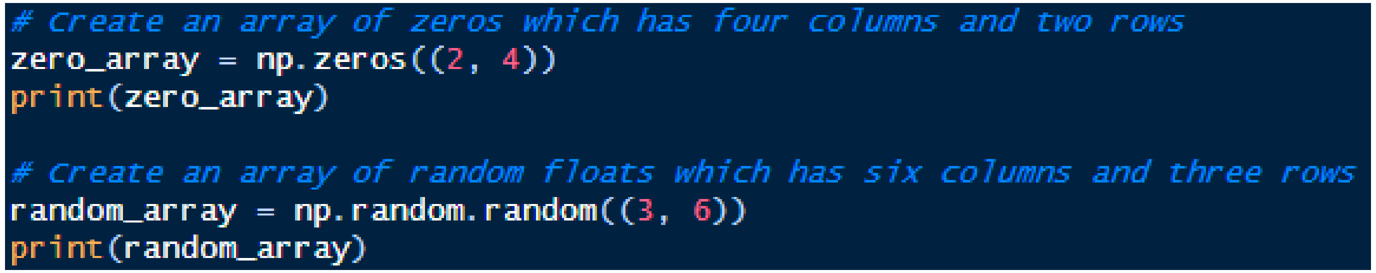


```
>>> # sudoku_list is a Python list containing a sudoku game
>>> sudoku_list =   [[0, 0, 4, 3, 0, 0, 2, 0, 9],
...                 [0, 0, 5, 0, 0, 9, 0, 0, 1],
...                 [0, 7, 0, 0, 6, 0, 0, 4, 3],
...                 [0, 0, 6, 0, 0, 2, 0, 8, 7],
...                 [1, 9, 0, 0, 0, 7, 4, 0, 0],
...                 [0, 5, 0, 0, 8, 3, 0, 0, 0],
...                 [6, 0, 0, 0, 0, 0, 1, 0, 5],
...                 [0, 0, 3, 5, 0, 8, 6, 9, 0],
...                 [0, 4, 2, 9, 1, 0, 3, 0, 0]]
>>>
>>>
>>> # Import NumPy
>>> import numpy as np
>>>
>>> # Convert sudoku_list into an array
>>> sudoku_array = np.array(sudoku_list)
>>>
>>> # Print the type of sudoku_array
>>> print(type(sudoku_array))
<class 'numpy.ndarray'>
```

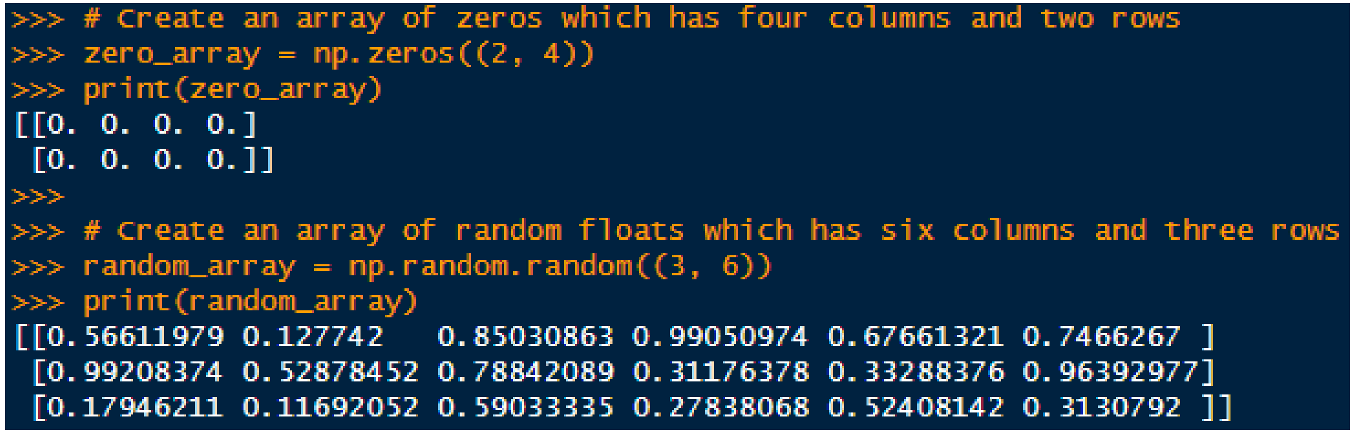
Source: Introduction to NumPy. Izzy Weber. Rpubs

Let’s Create Our Own NP Array From Scratch

1. We create and print an array filled with zeros called *zero_array*, which has two rows and four columns.
2. We create and print an array of random floats between *0* and *1* called *random_array*, which has three rows and six columns.



```
# Create an array of zeros which has four columns and two rows
zero_array = np.zeros((2, 4))
print(zero_array)

# Create an array of random floats which has six columns and three rows
random_array = np.random.random((3, 6))
print(random_array)
```

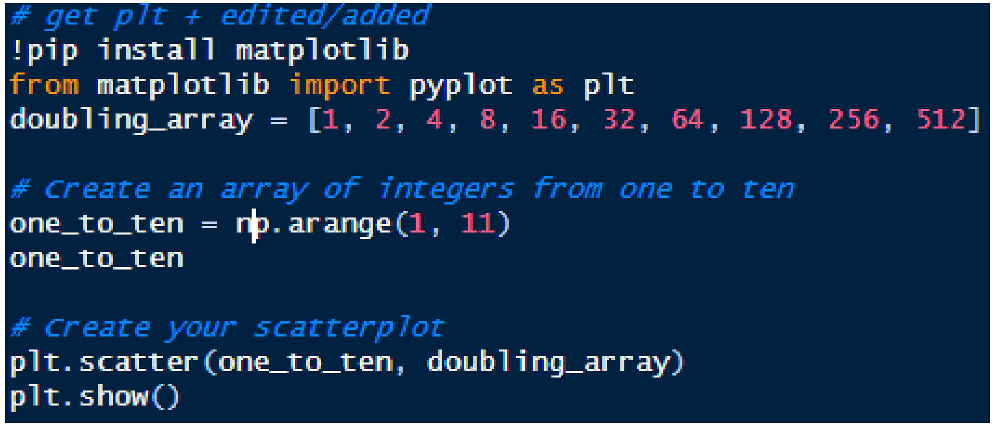


```
>>> # Create an array of zeros which has four columns and two rows
>>> zero_array = np.zeros((2, 4))
>>> print(zero_array)
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
>>>
>>> # Create an array of random floats which has six columns and three rows
>>> random_array = np.random.random((3, 6))
>>> print(random_array)
[[0.56611979 0.127742   0.85030863 0.99050974 0.67661321 0.7466267 ]
 [0.99208374 0.52878452 0.78842089 0.31176378 0.33288376 0.96392977]
 [0.17946211 0.11692052 0.59033335 0.27838068 0.52408142 0.3130792 ]]
```

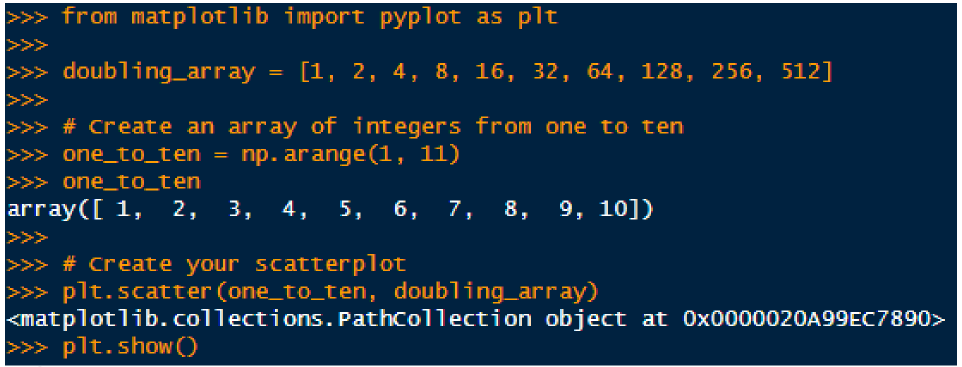
Source: Introduction to NumPy. Izzy Weber. Rpubs

Let’s Create a Range Array

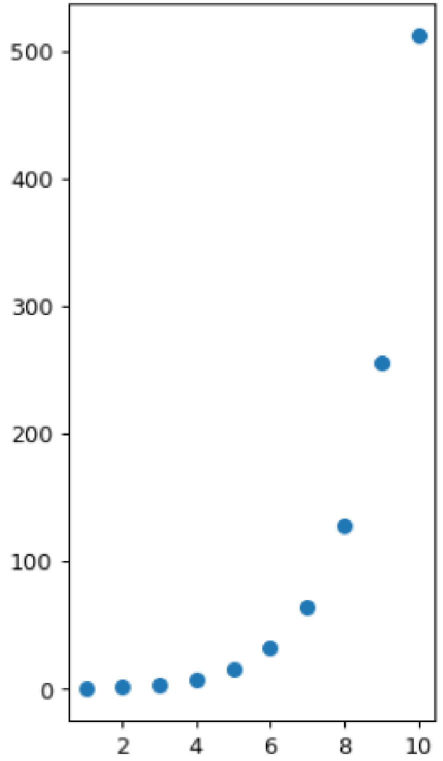
1. np.arange() has especially useful applications in graphing.
2. We create a scatter plot with the values from doubling_array on the y-axis, but we need values for the x-axis, which we can create with *np.arange()*.



```
# get plt + edited/added
!pip install matplotlib
from matplotlib import pyplot as plt
doubling_array = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

# Create an array of integers from one to ten
one_to_ten = np.arange(1, 11)
one_to_ten

# Create your scatterplot
plt.scatter(one_to_ten, doubling_array)
plt.show()
```



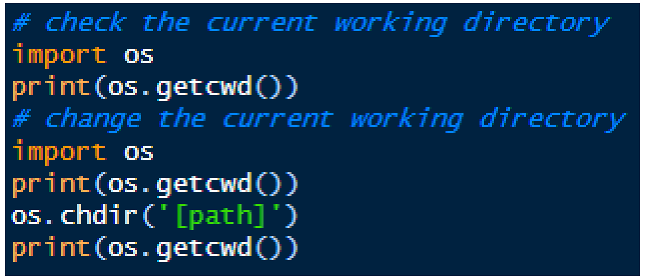
```
>>> from matplotlib import pyplot as plt
>>>
>>> doubling_array = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>>
>>> # Create an array of integers from one to ten
>>> one_to_ten = np.arange(1, 11)
>>> one_to_ten
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>>
>>> # Create your scatterplot
>>> plt.scatter(one_to_ten, doubling_array)
<matplotlib.collections.PathCollection object at 0x0000020A99EC7890>
>>> plt.show()
```



Source: Introduction to NumPy. Izzy Weber. Rpubs

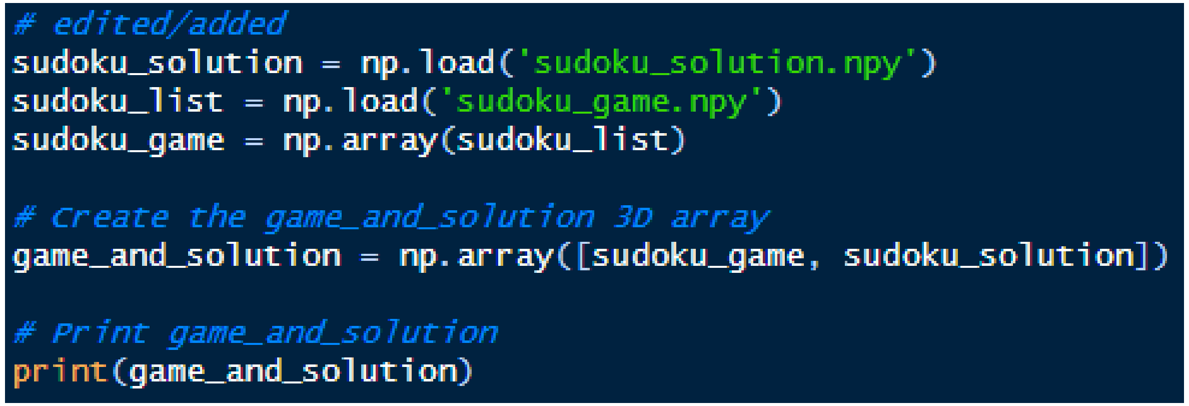
Array Dimensionality: 3D

Recall the *sudoku_array*. It’s actually 2D array. Now, let’s assume we have many games and we want to add the solution of the games. Hence, we add one more dimension to get 3D array.

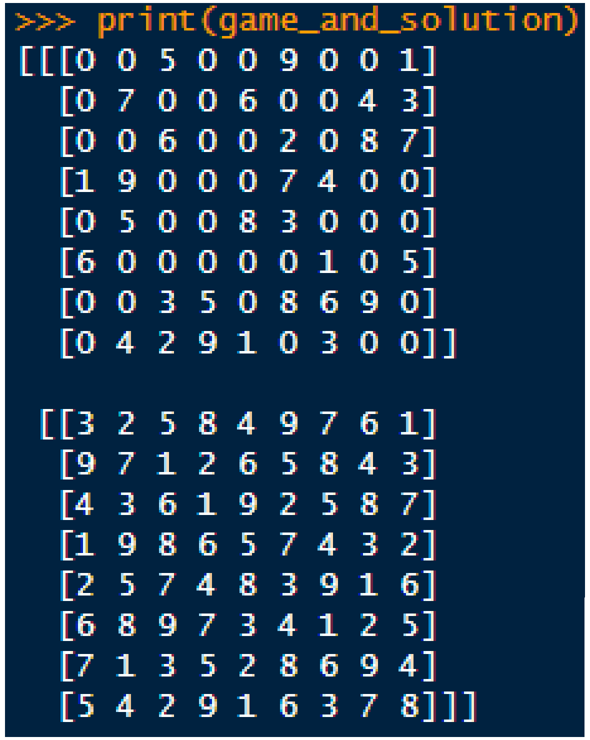


```
# check the current working directory
import os
print(os.getcwd())
# change the current working directory
import os
print(os.getcwd())
os.chdir('[path]')
print(os.getcwd())
```

```
For Google Colab:
1. Upload the file.
2.
import numpy as np
np.load('/tree_census.npy')
```



```
# edited/added
sudoku_solution = np.load('sudoku_solution.npy')
sudoku_list = np.load('sudoku_game.npy')
sudoku_game = np.array(sudoku_list)

# Create the game_and_solution 3D array
game_and_solution = np.array([sudoku_game, sudoku_solution])

# Print game_and_solution
print(game_and_solution)
```

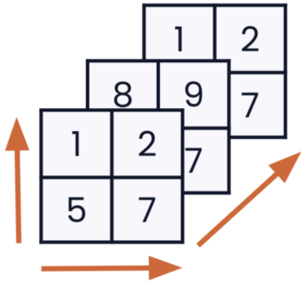


```
>>> print(game_and_solution)
[[[0 0 5 0 0 9 0 0 1]
  [0 7 0 0 6 0 0 4 3]
  [0 0 6 0 0 2 0 8 7]
  [1 9 0 0 0 7 4 0 0]
  [0 5 0 0 8 3 0 0 0]
  [6 0 0 0 0 0 1 0 5]
  [0 0 3 5 0 8 6 9 0]
  [0 4 2 9 1 0 3 0 0]]

 [[3 2 5 8 4 9 7 6 1]
  [9 7 1 2 6 5 8 4 3]
  [4 3 6 1 9 2 5 8 7]
  [1 9 8 6 5 7 4 3 2]
  [2 5 7 4 8 3 9 1 6]
  [6 8 9 7 3 4 1 2 5]
  [7 1 3 5 2 8 6 9 4]
  [5 4 2 9 1 6 3 7 8]]]
```

Explanation: We have two 2D arrays, 8 Rows per array, and 9 Columns.

Note: Whenever you see a "*Set of numbers*" closed in double brackets from both ends. Consider it as a "*set*". And 3D and 3D+ arrays are always built on these "sets".

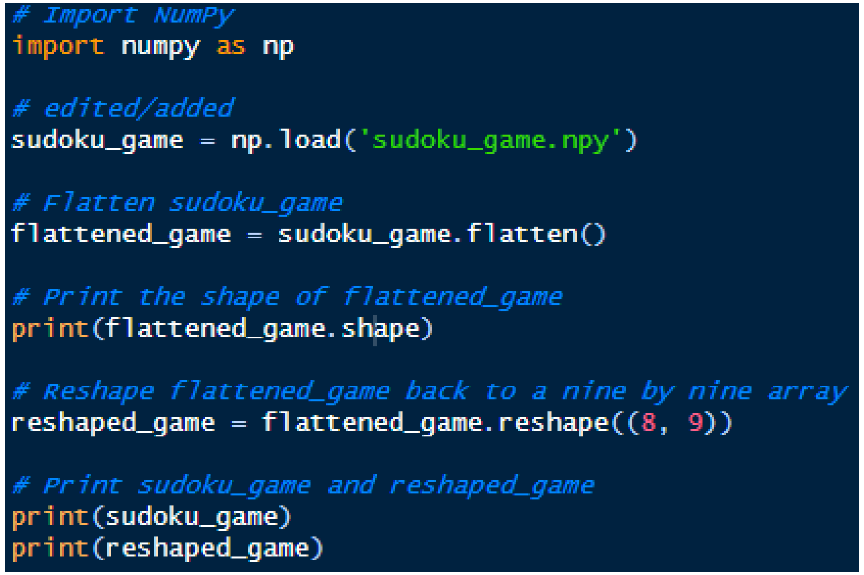
Source: Introduction to NumPy. Izzy Weber. Rpubs



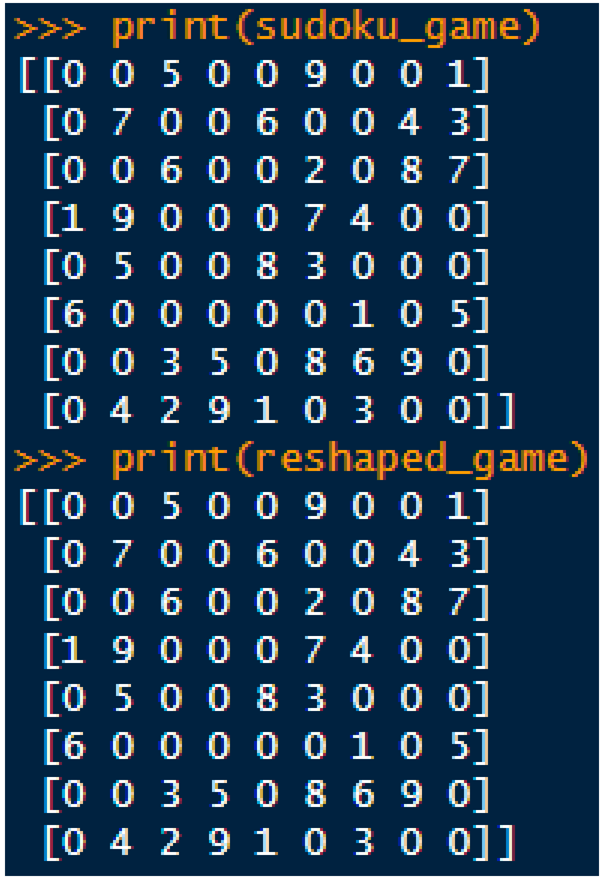
Flattening and Reshaping

We can change not only array shape but also the number of dimensions that an array has.

We change *sudoku_array* from a 2D array to a 1D array and back again.



```
# Import NumPy
import numpy as np

# edited/added
sudoku_game = np.load('sudoku_game.npy')

# Flatten sudoku_game
flattened_game = sudoku_game.flatten()

# Print the shape of flattened_game
print(flattened_game.shape)

# Reshape flattened_game back to a nine by nine array
reshaped_game = flattened_game.reshape((8, 9))

# Print sudoku_game and reshaped_game
print(sudoku_game)
print(reshaped_game)
```



```
>>> print(sudoku_game)
[[0 0 5 0 0 9 0 0 1]
 [0 7 0 0 6 0 0 4 3]
 [0 0 6 0 0 2 0 8 7]
 [1 9 0 0 0 7 4 0 0]
 [0 5 0 0 8 3 0 0 0]
 [6 0 0 0 0 0 1 0 5]
 [0 0 3 5 0 8 6 9 0]
 [0 4 2 9 1 0 3 0 0]]
>>> print(reshaped_game)
[[0 0 5 0 0 9 0 0 1]
 [0 7 0 0 6 0 0 4 3]
 [0 0 6 0 0 2 0 8 7]
 [1 9 0 0 0 7 4 0 0]
 [0 5 0 0 8 3 0 0 0]
 [6 0 0 0 0 0 1 0 5]
 [0 0 3 5 0 8 6 9 0]
 [0 4 2 9 1 0 3 0 0]]
```

Explanation:

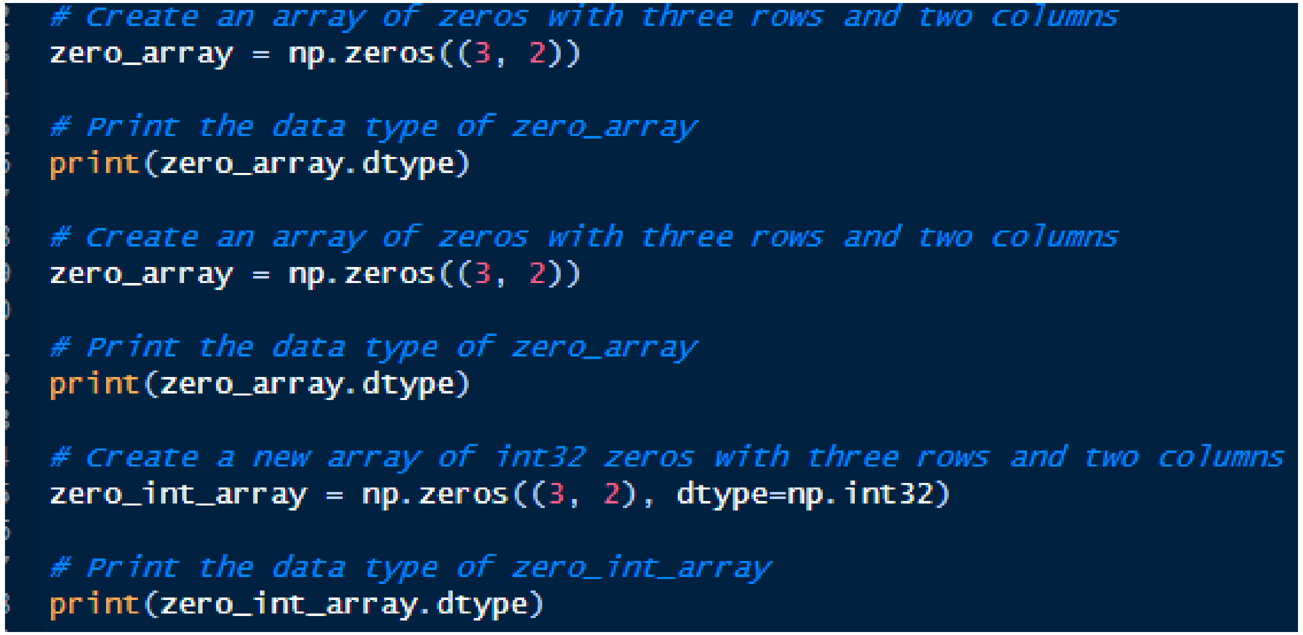
- Flatten *sudoku_game* so that it is a 1D array, and save it as *flattened_game*.
- *Print* the .shape of *flattened_game*.
- *Reshape* the *flattened_game* back to its original shape of nine rows and nine columns; save the new array as *reshaped_game*.

Source: Introduction to NumPy. Izzy Weber. Rpubs

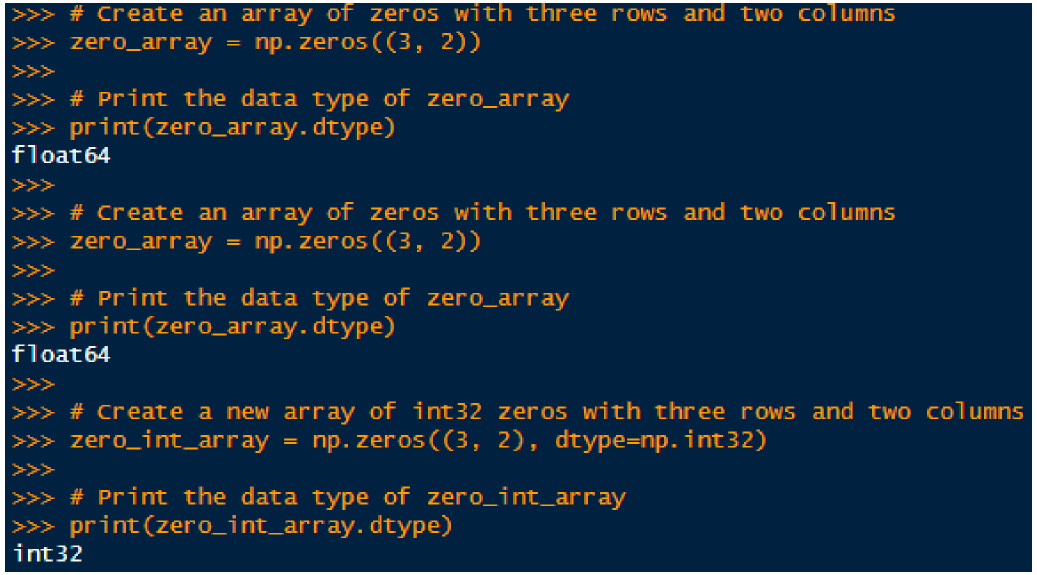
5. NumPy data types

The dtype Argument

One way to control the data type of a NumPy array is to declare it when the array is created using the *dtype* keyword argument.



```
# Create an array of zeros with three rows and two columns
zero_array = np.zeros((3, 2))

# Print the data type of zero_array
print(zero_array.dtype)

# Create an array of zeros with three rows and two columns
zero_array = np.zeros((3, 2))

# Print the data type of zero_array
print(zero_array.dtype)

# Create a new array of int32 zeros with three rows and two columns
zero_int_array = np.zeros((3, 2), dtype=np.int32)

# Print the data type of zero_int_array
print(zero_int_array.dtype)
```

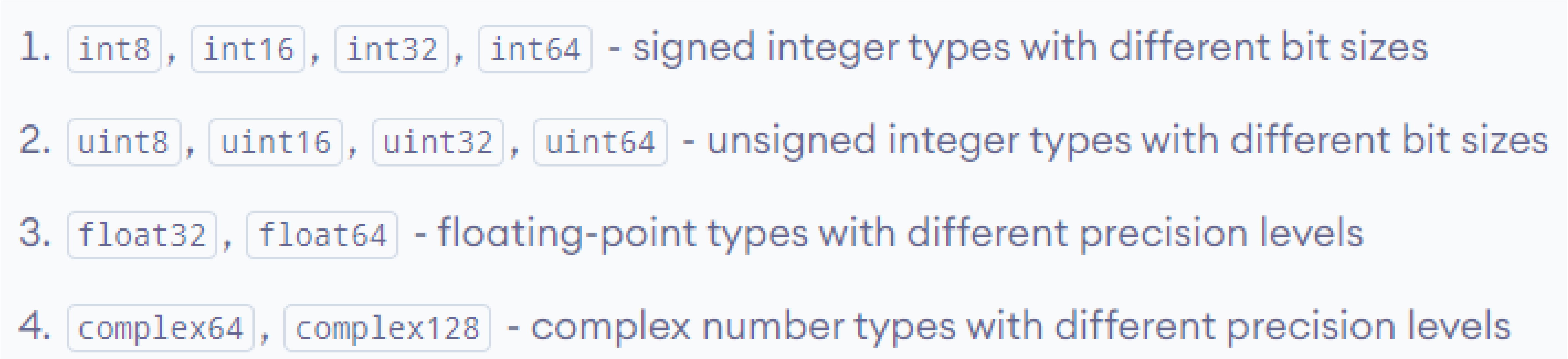


```
>>> # Create an array of zeros with three rows and two columns
>>> zero_array = np.zeros((3, 2))
>>>
>>> # Print the data type of zero_array
>>> print(zero_array.dtype)
float64
>>>
>>> # Create an array of zeros with three rows and two columns
>>> zero_array = np.zeros((3, 2))
>>>
>>> # Print the data type of zero_array
>>> print(zero_array.dtype)
float64
>>>
>>> # Create a new array of int32 zeros with three rows and two columns
>>> zero_int_array = np.zeros((3, 2), dtype=np.int32)
>>>
>>> # Print the data type of zero_int_array
>>> print(zero_int_array.dtype)
int32
```

Source: Introduction to NumPy. Izzy Weber. Rpubs

Numpy Data Types

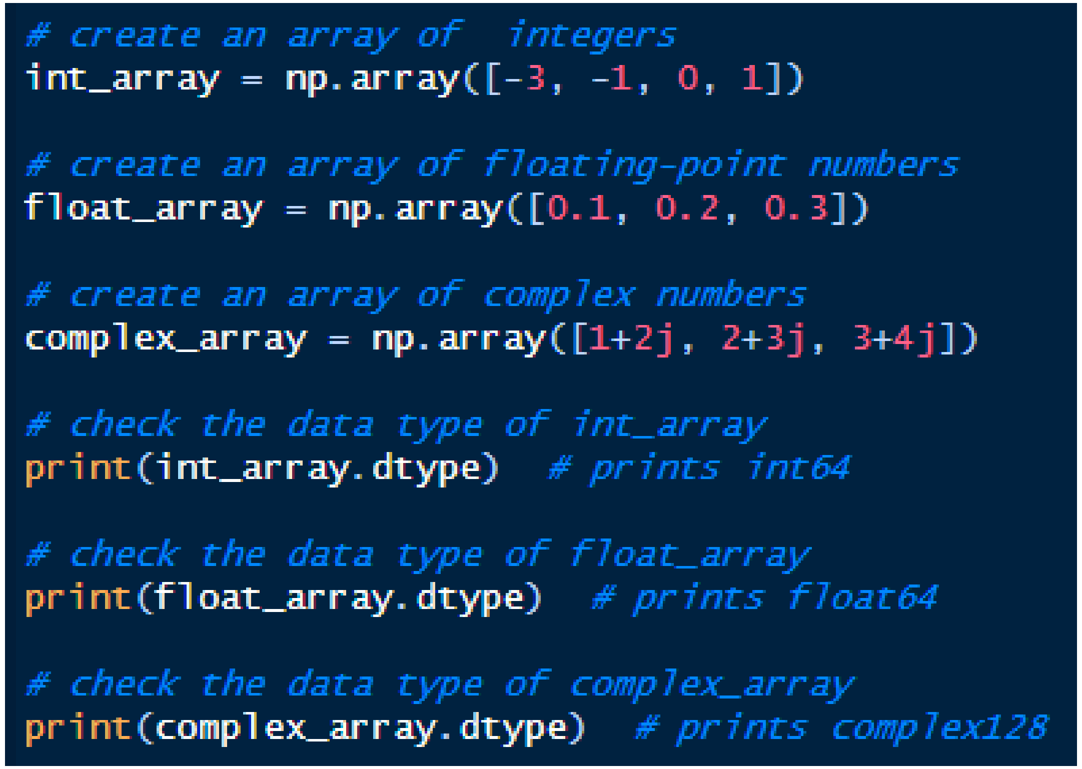
NumPy offers a wider range of numerical data types than what is available in Python. Here's the list of most commonly used numeric data types in NumPy:



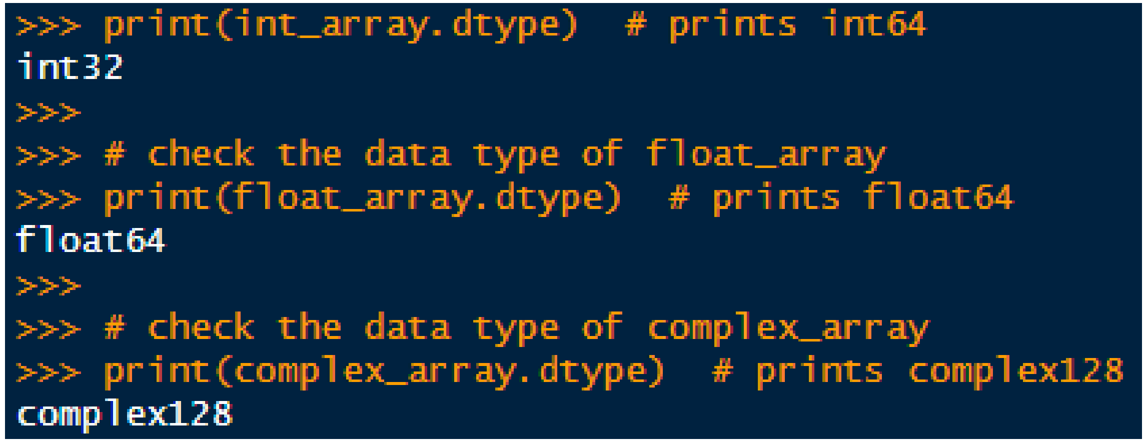
1. `int8`, `int16`, `int32`, `int64` - signed integer types with different bit sizes
2. `uint8`, `uint16`, `uint32`, `uint64` - unsigned integer types with different bit sizes
3. `float32`, `float64` - floating-point types with different precision levels
4. `complex64`, `complex128` - complex number types with different precision levels

Source: Learn NumPy Interactively. Programiz.

Example: Check Data Type of NumPy Array



```python
# create an array of  integers
int_array = np.array([-3, -1, 0, 1])

# create an array of floating-point numbers
float_array = np.array([0.1, 0.2, 0.3])

# create an array of complex numbers
complex_array = np.array([1+2j, 2+3j, 3+4j])

# check the data type of int_array
print(int_array.dtype)  # prints int64

# check the data type of float_array
print(float_array.dtype)  # prints float64

# check the data type of complex_array
print(complex_array.dtype)  # prints complex128
```

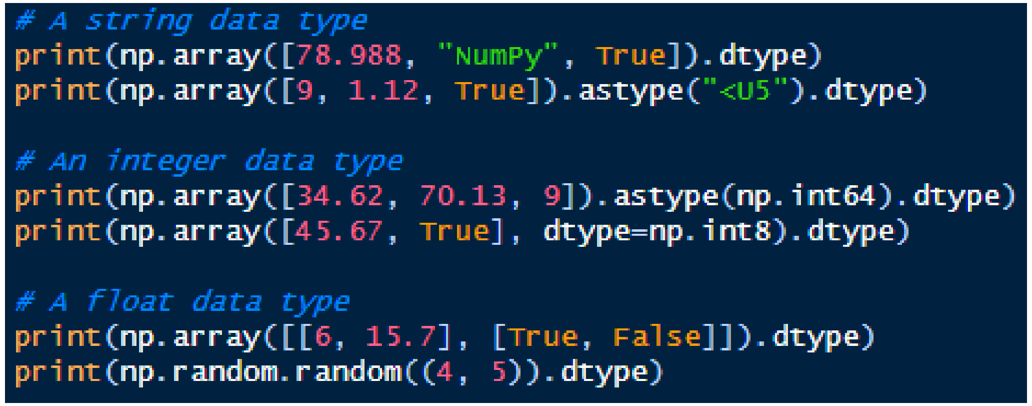


```
>>> print(int_array.dtype)  # prints int64
int32
>>>
>>> # check the data type of float_array
>>> print(float_array.dtype)  # prints float64
float64
>>>
>>> # check the data type of complex_array
>>> print(complex_array.dtype)  # prints complex128
complex128
```

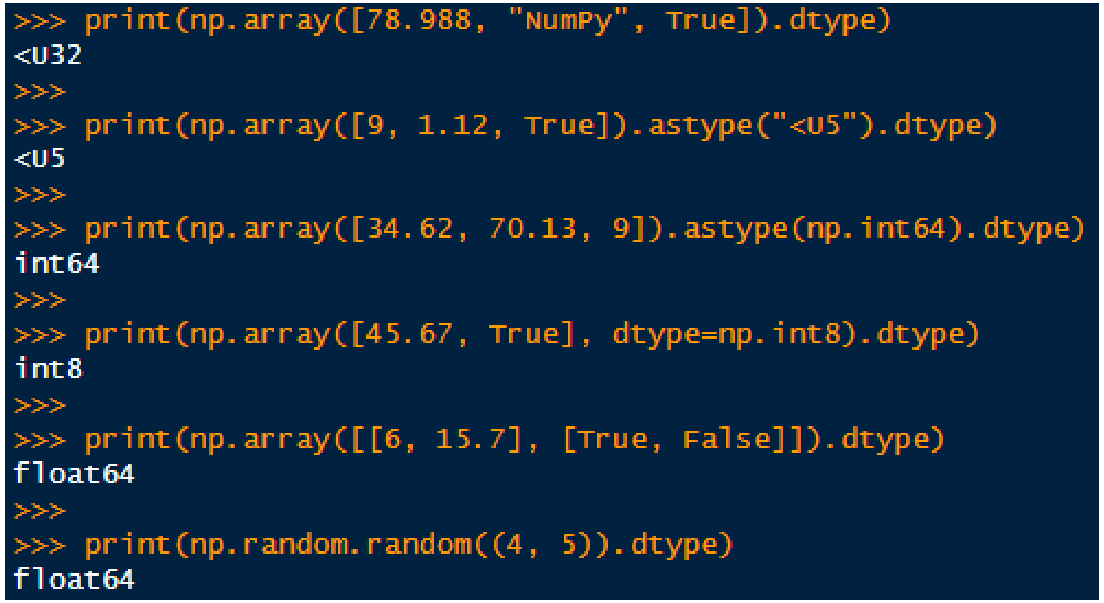
Source: Learn NumPy Interactively. Programiz.

Anticipating Data Types

Anticipating what data type an array will have is very important since some NumPy functionality only works with certain data types.



```
# A string data type
print(np.array([78.988, "NumPy", True]).dtype)
print(np.array([9, 1.12, True]).astype("<U5").dtype)

# An integer data type
print(np.array([34.62, 70.13, 9]).astype(np.int64).dtype)
print(np.array([45.67, True], dtype=np.int8).dtype)

# A float data type
print(np.array([[6, 15.7], [True, False]]).dtype)
print(np.random.random((4, 5)).dtype)
```



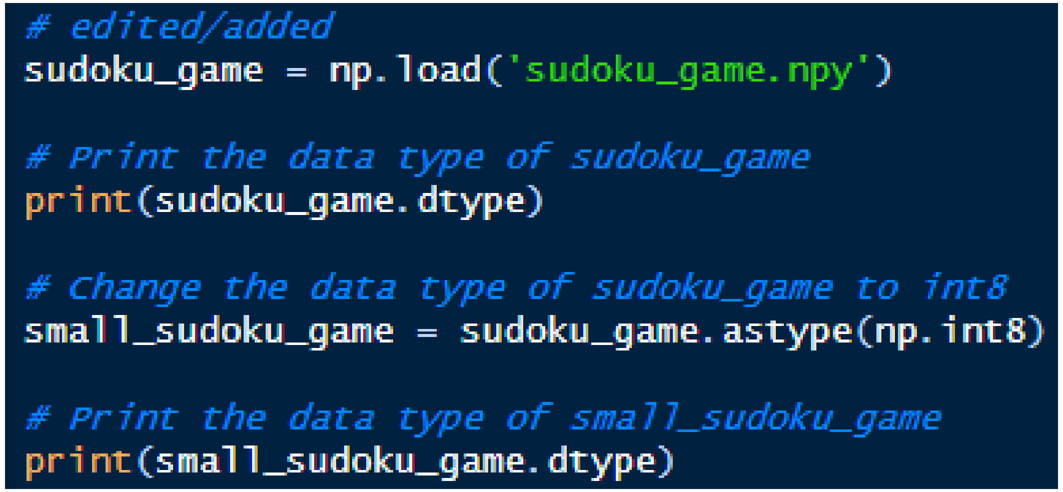
```
>>> print(np.array([78.988, "NumPy", True]).dtype)
<U32
>>>
>>> print(np.array([9, 1.12, True]).astype("<U5").dtype)
<U5
>>>
>>> print(np.array([34.62, 70.13, 9]).astype(np.int64).dtype)
int64
>>>
>>> print(np.array([45.67, True], dtype=np.int8).dtype)
int8
>>>
>>> print(np.array([[6, 15.7], [True, False]]).dtype)
float64
>>>
>>> print(np.random.random((4, 5)).dtype)
float64
```

Source: Introduction to NumPy. Izzy Weber. Rpubs

Change Data Type

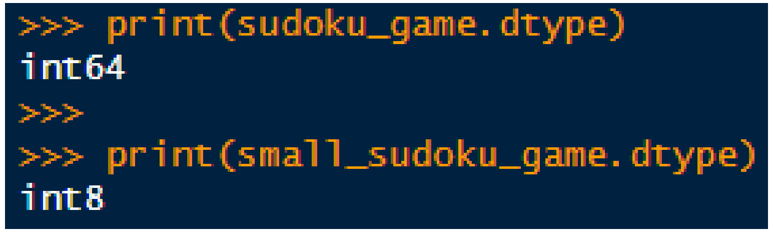
NumPy data types, which emphasize speed, are more specific than Python data types, which emphasize flexibility.

When working with large amounts of data in NumPy, it’s good practice to check the data type and consider whether a smaller data type is large enough for your data, since smaller data types use less memory.



```
# edited/added
sudoku_game = np.load('sudoku_game.npy')

# Print the data type of sudoku_game
print(sudoku_game.dtype)

# Change the data type of sudoku_game to int8
small_sudoku_game = sudoku_game.astype(np.int8)

# Print the data type of small_sudoku_game
print(small_sudoku_game.dtype)
```

We make *sudoku game* more memory-efficient.



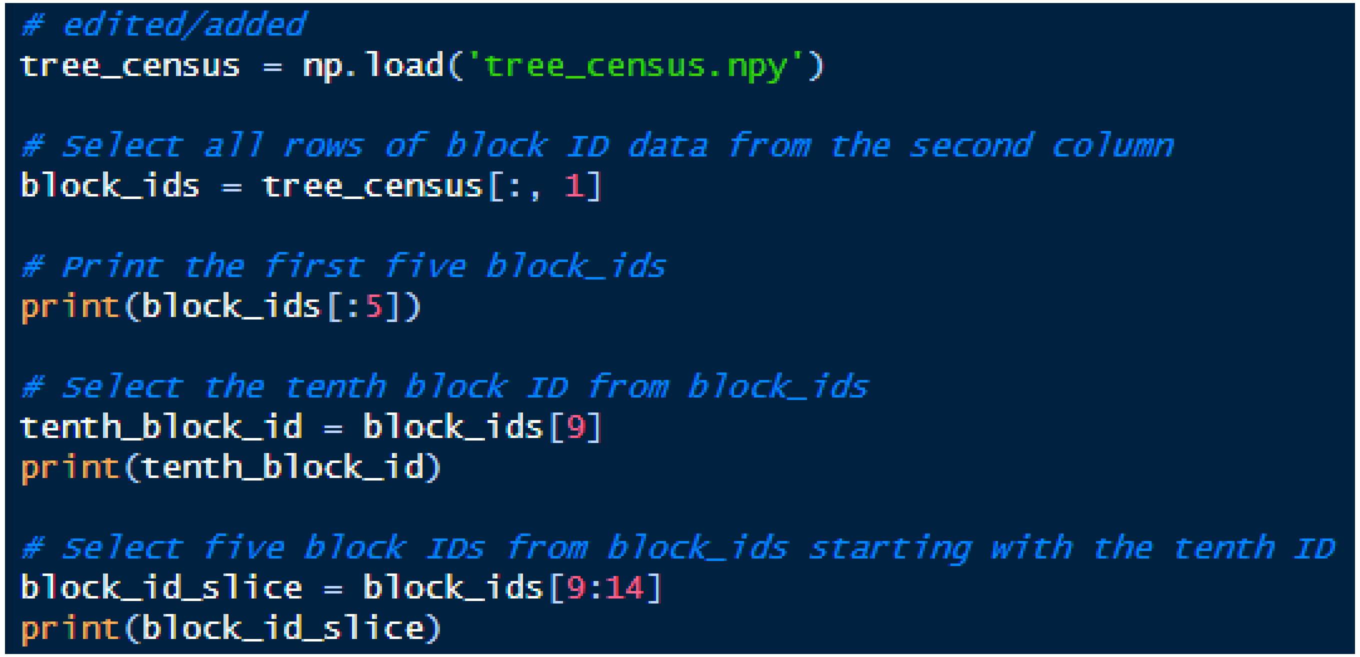
```
>>> print(sudoku_game.dtype)
int64
>>>
>>> print(small_sudoku_game.dtype)
int8
```

Source: Introduction to NumPy. Izzy Weber. Rpubs

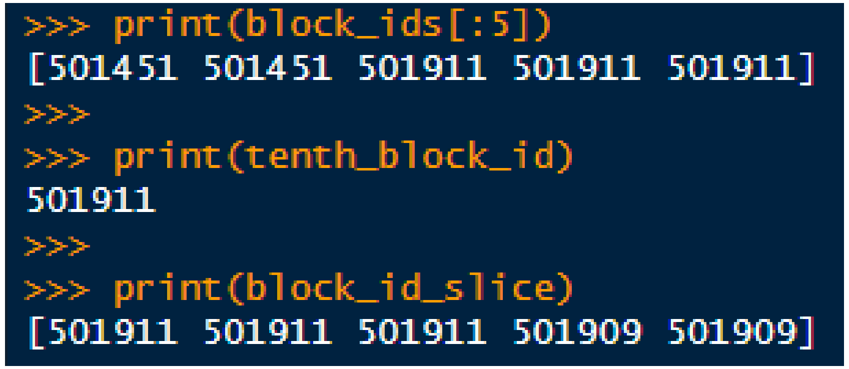
6. Selecting and Updating Data

Slicing and Indexing Trees

- We’ll be working specifically with the second column, representing block IDs.
- We will select specific city blocks for further analysis using NumPy slicing and indexing.



```
# edited/added
tree_census = np.load('tree_census.npy')

# Select all rows of block ID data from the second column
block_ids = tree_census[:, 1]

# Print the first five block_ids
print(block_ids[:5])

# Select the tenth block ID from block_ids
tenth_block_id = block_ids[9]
print(tenth_block_id)

# Select five block IDs from block_ids starting with the tenth ID
block_id_slice = block_ids[9:14]
print(block_id_slice)
```

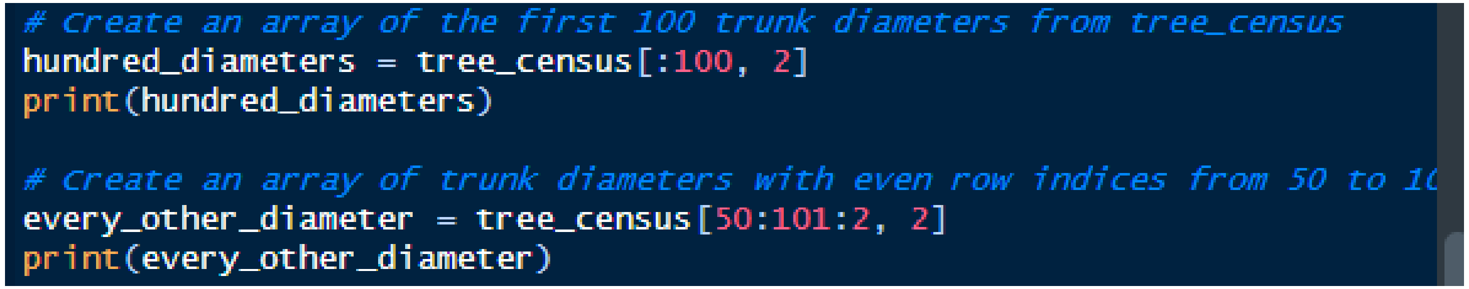


```
>>> print(block_ids[:5])
[501451 501451 501911 501911 501911]
>>>
>>> print(tenth_block_id)
501911
>>>
>>> print(block_id_slice)
[501911 501911 501911 501909 501909]
```

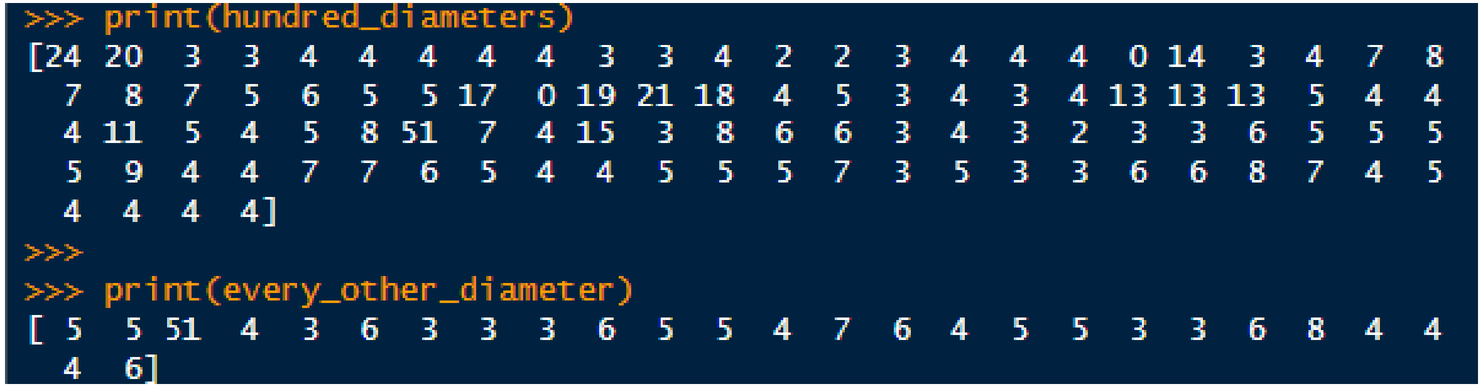
Source: Introduction to NumPy. Izzy Weber. Rpubs

Slicing and Indexing Trees

- We create an array called *hundred_diameters* which contains the first 100 trunk diameters (from column 3) in *tree_census*.
- We create an array, *every_other_diameter*, which contains only trunk diameters for trees with even row indices from 50 to 100, inclusive.



```
# Create an array of the first 100 trunk diameters from tree_census
hundred_diameters = tree_census[:100, 2]
print(hundred_diameters)

# Create an array of trunk diameters with even row indices from 50 to 10
every_other_diameter = tree_census[50:101:2, 2]
print(every_other_diameter)
```

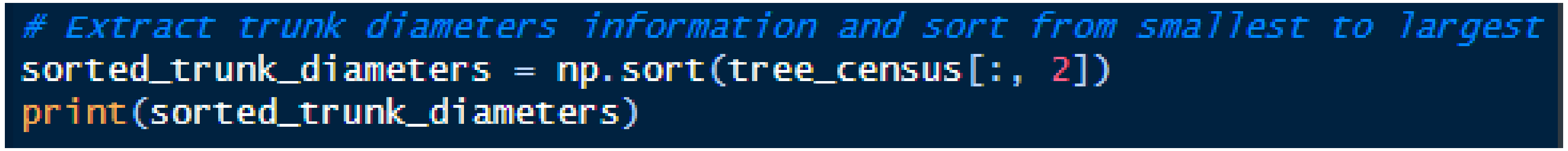


```
>>> print(hundred_diameters)
[24 20  3  3  4  4  4  4  4  3  3  4  2  2  3  4  4  4  0 14  3  4  7  8
  7  8  7  5  6  5  5 17  0 19 21 18  4  5  3  4  3  4 13 13 13  5  4  4
  4 11  5  4  5  8 51  7  4 15  3  8  6  6  3  4  3  2  3  3  6  5  5  5
  5  9  4  4  7  7  6  5  4  4  5  5  5  7  3  5  3  3  6  6  8  7  4  5
  4  4  4  4]
>>>
>>> print(every_other_diameter)
[ 5  5 51  4  3  6  3  3  3  6  5  5  4  7  6  4  5  5  3  3  6  8  4  4
  4  6]
```

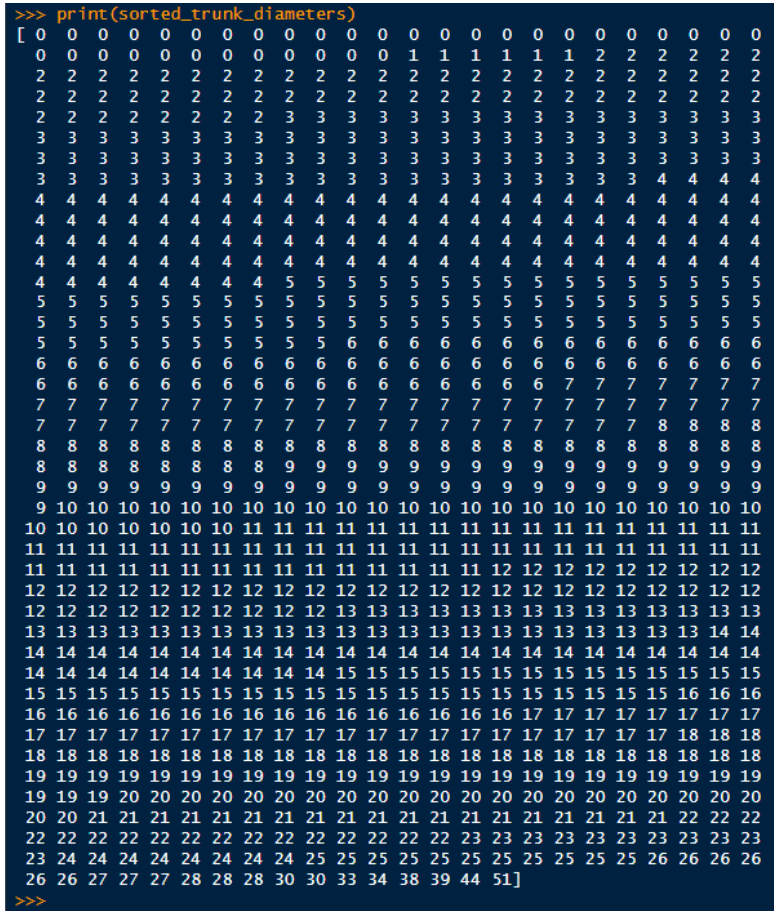
Source: Introduction to NumPy. Izzy Weber. Rpubs

Sorting Trees

- Create an array called *sorted_trunk_diameters* which selects only the trunk diameter column from *tree_census* and sorts it so that the smallest trunk diameters are at the top of the array and the largest at the bottom.



```
# Extract trunk diameters information and sort from smallest to largest
sorted_trunk_diameters = np.sort(tree_census[:, 2])
print(sorted_trunk_diameters)
```

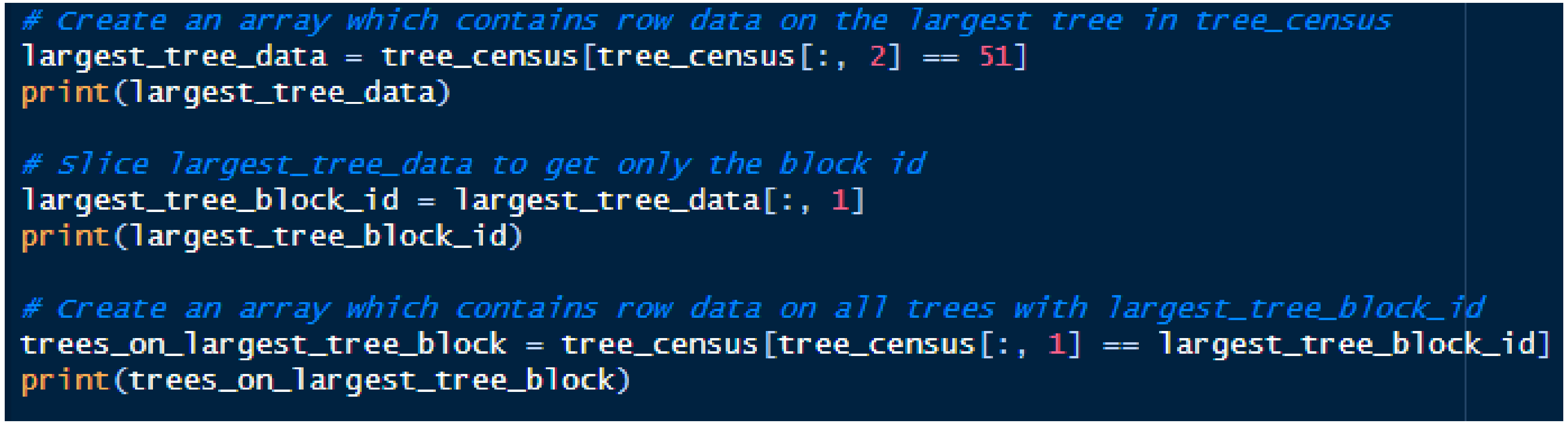


```
>>> print(sorted_trunk_diameters)
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  2  2  2  2  2  2
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  4  4  4  4
  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  6  6  6  6  6  6  6  6  6  6  6  6  6  6
  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6
  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7
  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7
  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
  9 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 12 12 12 12 12 12 12 12 12
 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 13 13 13 13
 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 14 14
 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
 14 14 14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15 15 15 15 15 15 15
 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 16 16 16
 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 18 18 18
 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
 19 19 19 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 22 22 22
 22 22 22 22 22 22 22 22 22 22 22 22 22 22 23 23 23 23 23 23 23 23 23 23
 23 24 24 24 24 24 24 24 25 25 25 25 25 25 25 25 25 25 25 25 26 26 26 26
 26 26 27 27 27 28 28 28 30 30 33 34 38 39 44 51]
>>>
```

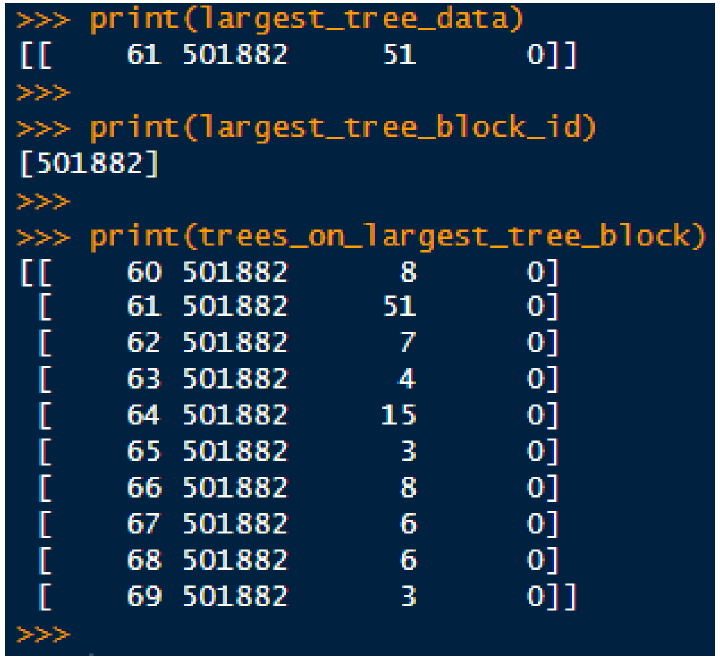
Source: Introduction to NumPy. Izzy Weber. Rpubs

Filtering with Masks

- We’ll use fancy indexing to return the row of data representing the largest tree in *tree_census*.
- Using Boolean indexing, we create an array, *largest_tree_data*, which contains the row of data on the largest tree in *tree_census* corresponding to the tree with a diameter of 51.



```
# Create an array which contains row data on the largest tree in tree_census
largest_tree_data = tree_census[tree_census[:, 2] == 51]
print(largest_tree_data)

# Slice largest_tree_data to get only the block id
largest_tree_block_id = largest_tree_data[:, 1]
print(largest_tree_block_id)

# Create an array which contains row data on all trees with largest_tree_block_id
trees_on_largest_tree_block = tree_census[tree_census[:, 1] == largest_tree_block_id]
print(trees_on_largest_tree_block)
```

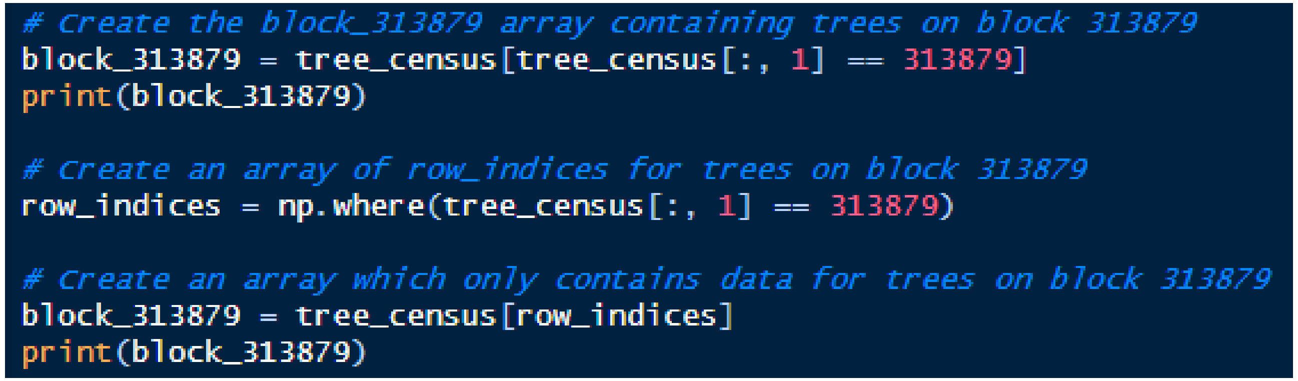


```
>>> print(largest_tree_data)
[[    61 501882     51      0]]
>>>
>>> print(largest_tree_block_id)
[501882]
>>>
>>> print(trees_on_largest_tree_block)
[[    60 501882      8      0]
 [    61 501882     51      0]
 [    62 501882      7      0]
 [    63 501882      4      0]
 [    64 501882     15      0]
 [    65 501882      3      0]
 [    66 501882      8      0]
 [    67 501882      6      0]
 [    68 501882      6      0]
 [    69 501882      3      0]]
>>>
```

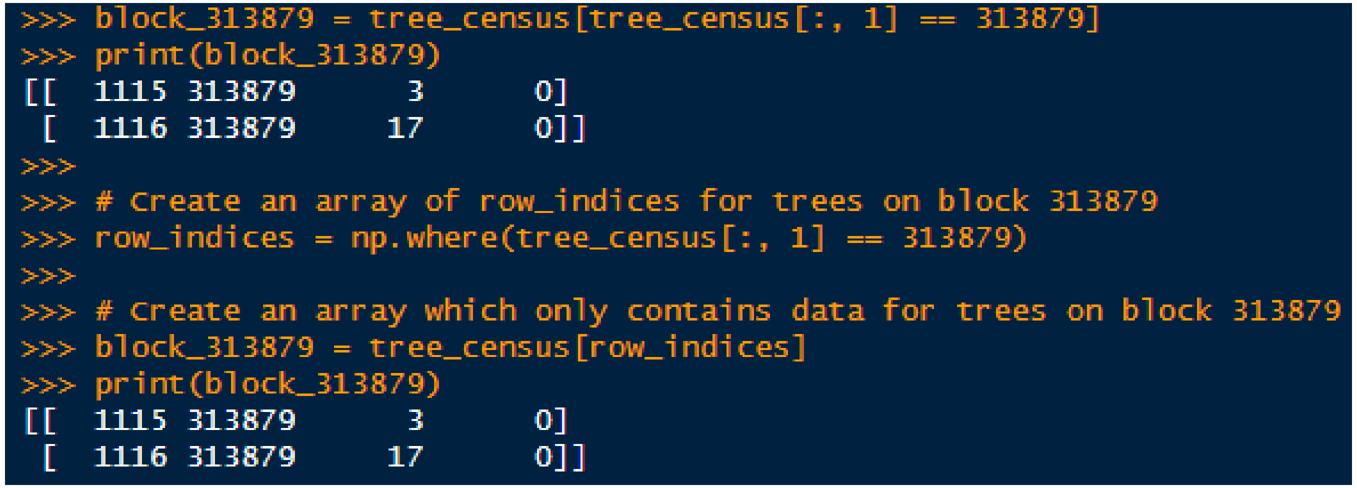
Source: Introduction to NumPy. Izzy Weber. Rpubs

Fancy Indexing vs. np.where()

- You’ve been assigned to check the data for trees on block *313879*, and you’d like to make a small array of just the tree data that relates to your work.



```
# Create the block_313879 array containing trees on block 313879
block_313879 = tree_census[tree_census[:, 1] == 313879]
print(block_313879)

# Create an array of row_indices for trees on block 313879
row_indices = np.where(tree_census[:, 1] == 313879)

# Create an array which only contains data for trees on block 313879
block_313879 = tree_census[row_indices]
print(block_313879)
```

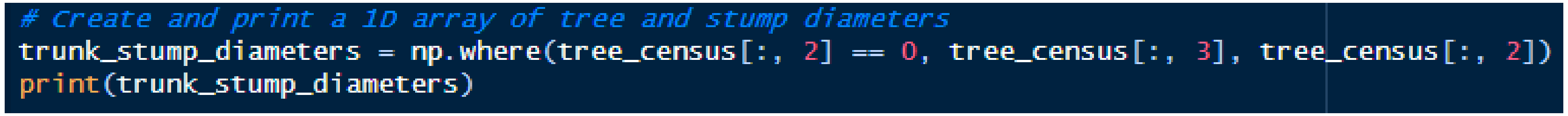


```
>>> block_313879 = tree_census[tree_census[:, 1] == 313879]
>>> print(block_313879)
[[   1115 313879      3      0]
 [   1116 313879     17      0]]
>>>
>>> # Create an array of row_indices for trees on block 313879
>>> row_indices = np.where(tree_census[:, 1] == 313879)
>>>
>>> # Create an array which only contains data for trees on block 313879
>>> block_313879 = tree_census[row_indices]
>>> print(block_313879)
[[   1115 313879      3      0]
 [   1116 313879     17      0]]
```

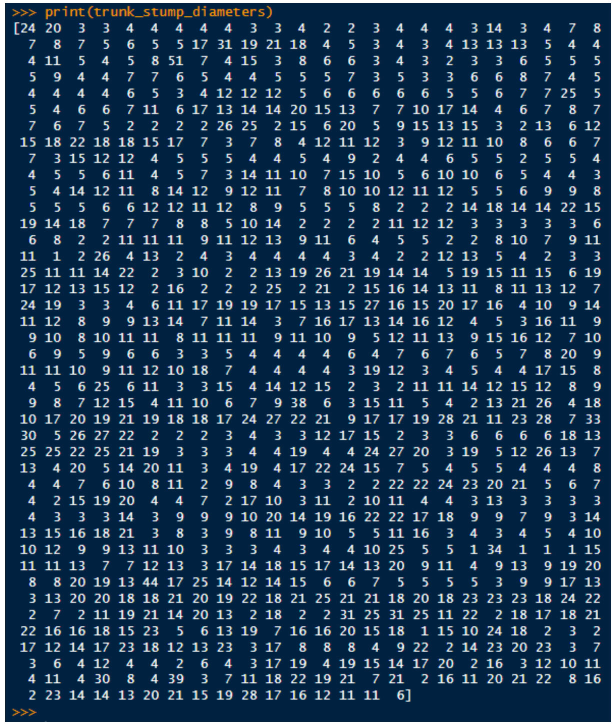
Source: Introduction to NumPy. Izzy Weber. Rpubs

Creating Arrays From Conditions

- Create and print a 1D array called *trunk_stump_diameters*, which replaces a tree's trunk diameter with its stump diameter if the trunk diameter is zero.



```
# Create and print a 1D array of tree and stump diameters
trunk_stump_diameters = np.where(tree_census[:, 2] == 0, tree_census[:, 3], tree_census[:, 2])
print(trunk_stump_diameters)
```

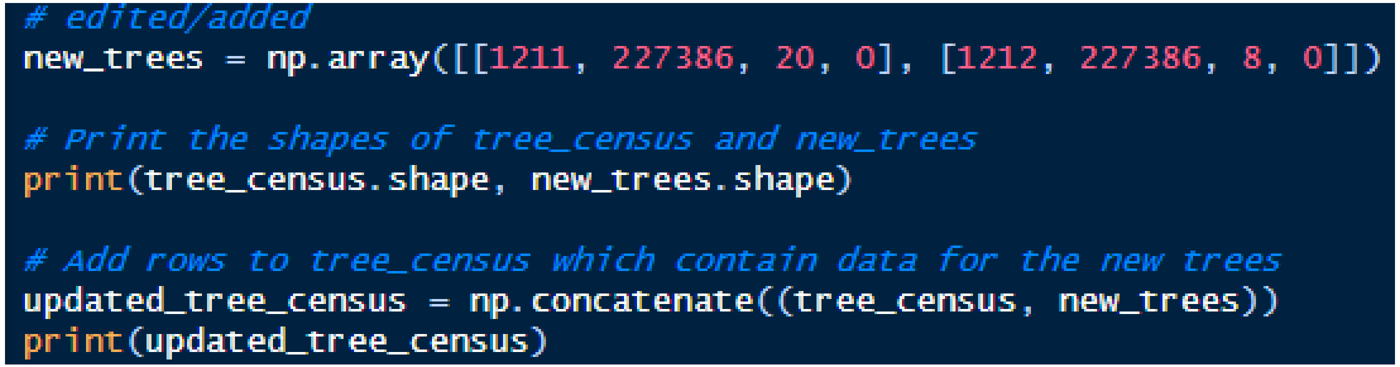


```
>>> print(trunk_stump_diameters)
[24 20  3  3  4  4  4  4  4  3  3  4  2  2  3  4  4  4  3 14  3  4  7  8
  7  8  7  5  6  5  5 17 31 19 21 18  4  5  3  4  3  4 13 13 13  5  4  4
  4 11  5  4  5  8 51  7  4 15  3  8  6  6  3  4  3  2  3  3  6  5  5  5
  5  9  4  4  7  7  6  5  4  4  5  5  5  7  3  5  3  3  6  6  8  7  4  5
  4  4  4  4  6  5  3  4 12 12 12  5  6  6  6  6  6  5  5  6  7  7 25  5
  5  4  6  6  7 11  6 17 13 14 14 20 15 13  7  7 10 17 14  4  6  7  8  7
  7  6  7  5  2  2  2  2 26 25  2 15  6 20  5  9 15 13 15  3  2 13  6 12
 15 18 22 18 18 15 17  7  3  7  8  4 12 11 12  3  9 12 11 10  8  6  6  7
  7  3 15 12 12  4  5  5  5  4  4  5  4  9  2  4  4  6  5  5  2  5  5  4
  4  5  5  6 11  4  5  7  3 14 11 10  7 15 10  5  6 10 10  6  5  4  4  3
  5  4 14 12 11  8 14 12  9 12 11  7  8 10 10 12 11 12  5  5  6  9  9  8
  5  5  5  6  6 12 12 11 12  8  9  5  5  5  8  2  2  2 14 18 14 14 22 15
 19 14 18  7  7  7  8  8  5 10 14  2  2  2  2 11 12 12  3  3  3  3  3  6
  6  8  2  2 11 11 11  9 11 12 13  9 11  6  4  5  5  2  2  8 10  7  9 11
 11  1  2 26  4 13  2  4  3  4  4  4  4  3  4  2  2 12 13  5  4  2  3  3
 25 11 11 14 22  2  3 10  2  2 13 19 26 21 19 14 14  5 19 15 11 15  6 19
 17 12 13 15 12  2 16  2  2  2 25  2 21  2 15 16 14 13 11  8 11 13 12  7
 24 19  3  3  4  6 11 17 19 19 17 15 13 15 27 16 15 20 17 16  4 10  9 14
 11 12  8  9  9 13 14  7 11 14  3  7 16 17 13 14 16 12  4  5  3 16 11  9
  9 10  8 10 11 11  8 11 11 11  9 11 10  9  5 12 11 13  9 15 16 12  7 10
  6  9  5  9  6  6  3  3  5  4  4  4  4  6  4  7  6  7  6  5  7  8 20  9
 11 11 10  9 11 12 10 18  7  4  4  4  4  3 19 12  3  4  5  4  4 17 15  8
  4  5  6 25  6 11  3  3 15  4 14 12 15  2  3  2 11 11 14 12 15 12  8  9
  9  8  7 12 15  4 11 10  6  7  9 38  6  3 15 11  5  4  2 13 21 26  4 18
 10 17 20 19 21 19 18 18 17 24 27 22 21  9 17 17 19 28 21 11 23 28  7 33
 30  5 26 27 22  2  2  2  3  4  3  3 12 17 15  2  3  3  6  6  6  6 18 13
 25 25 22 25 21 19  3  3  3  4  4 19  4  4 24 27 20  3 19  5 12 26 13  7
 13  4 20  5 14 20 11  3  4 19  4 17 22 24 15  7  5  4  5  5  4  4  4  8
  4  4  7  6 10  8 11  2  9  8  4  3  3  2  2 22 22 24 23 20 21  5  6  7
  4  2 15 19 20  4  4  7  2 17 10  3 11  2 10 11  4  4  3 13  3  3  3  3
  4  3  3  3 14  3  9  9  9 10 20 14 19 16 22 22 17 18  9  9  7  9  3 14
 13 15 16 18 21  3  8  3  9  8 11  9 10  5  5 11 16  3  4  3  4  5  4 10
 10 12  9  9 13 11 10  3  3  3  4  3  4  4 10 25  5  5  1 34  1  1  1 15
 11 11 13  7  7 12 13  3 17 14 18 15 17 14 13 20  9 11  4  9 13  9 19 20
  8  8 20 19 13 44 17 25 14 12 14 15  6  6  7  5  5  5  5  3  9  9 17 13
  3 13 20 20 18 18 21 20 19 22 18 21 25 21 21 18 20 18 23 23 23 18 24 22
  2  7  2 11 19 21 14 20 13  2 18  2  2 31 25 31 25 11 22  2 18 17 18 21
 22 16 16 18 15 23  5  6 13 19  7 16 16 20 15 18  1 15 10 24 18  2  3  2
 17 12 14 17 23 18 12 13 23  3 17  8  8  8  4  9 22  2 14 23 20 23  3  7
  3  6  4 12  4  4  2  6  4  3 17 19  4 19 15 14 17 20  2 16  3 12 10 11
  4 11  4 30  8  4 39  3  7 11 18 22 19 21  7 21  2 16 11 20 21 22  8 16
  2 23 14 14 13 20 21 15 19 28 17 16 12 11 11  6]
>>>
```

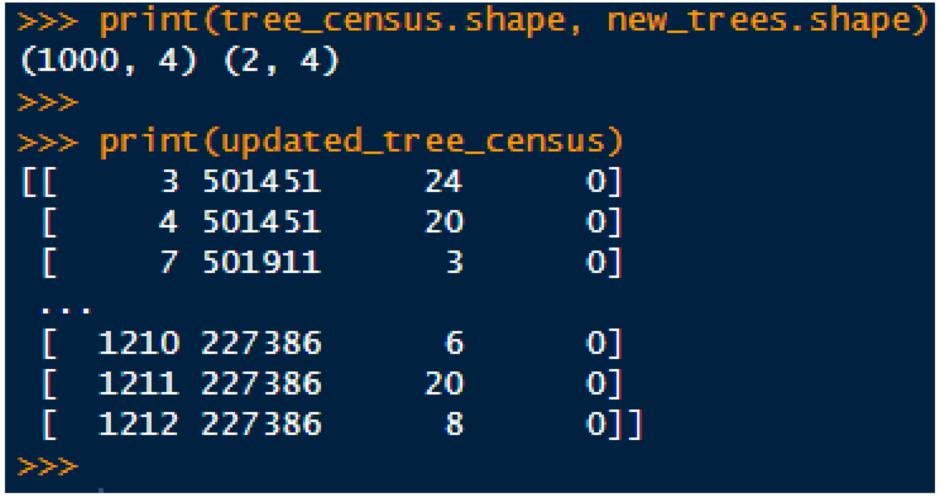
Source: Introduction to NumPy. Izzy Weber. Rpubs

Adding Rows

- We add rows containing the data for two new trees to the end of the *tree_census* array. The new trees' data is saved in a 2D array called *new_trees*



```
# edited/added
new_trees = np.array([[1211, 227386, 20, 0], [1212, 227386, 8, 0]])

# Print the shapes of tree_census and new_trees
print(tree_census.shape, new_trees.shape)

# Add rows to tree_census which contain data for the new trees
updated_tree_census = np.concatenate((tree_census, new_trees))
print(updated_tree_census)
```

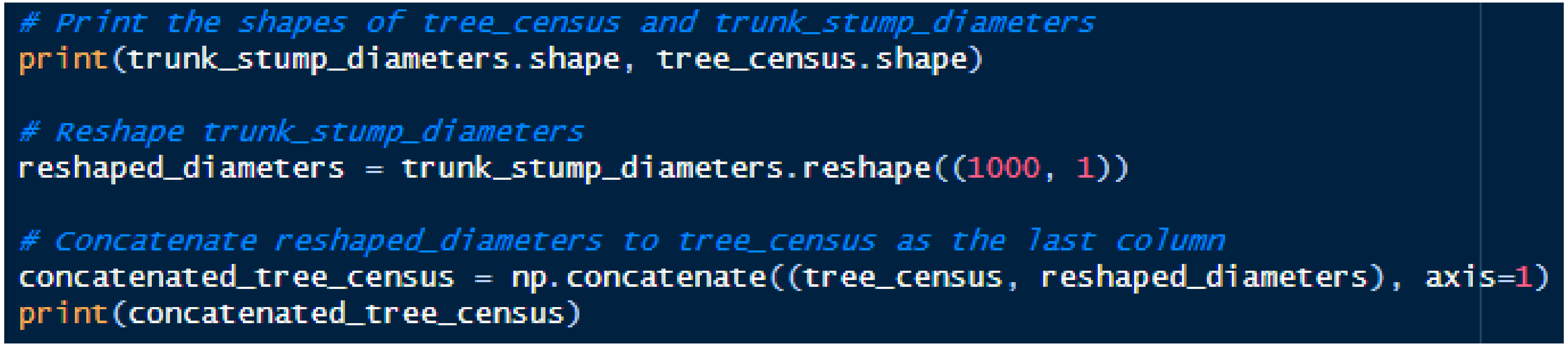


```
>>> print(tree_census.shape, new_trees.shape)
(1000, 4) (2, 4)
>>>
>>> print(updated_tree_census)
[[     3 501451     24      0]
 [     4 501451     20      0]
 [     7 501911      3      0]
 ...
 [  1210 227386      6      0]
 [  1211 227386     20      0]
 [  1212 227386      8      0]]
>>>
```

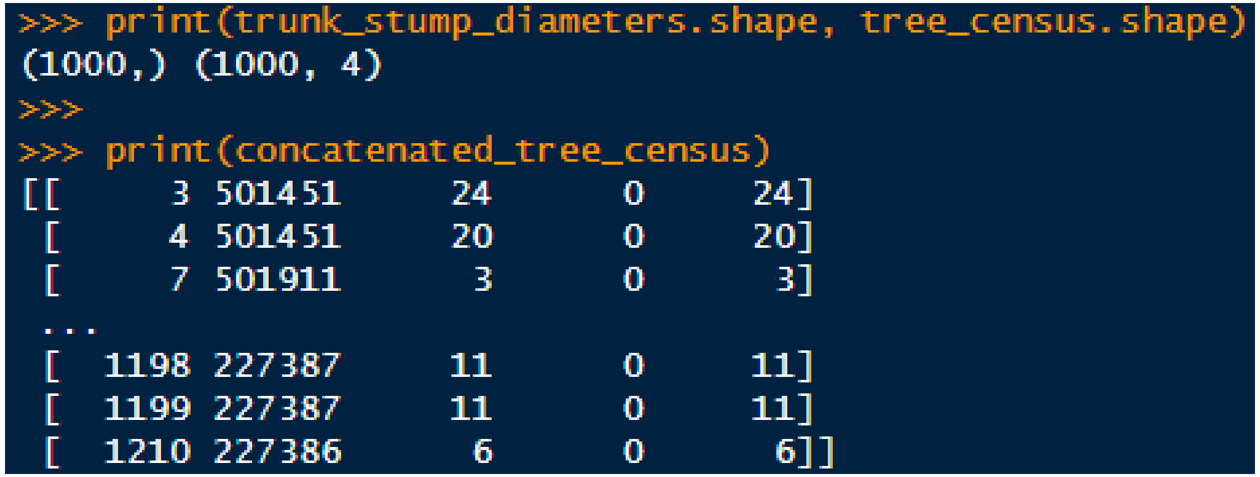
Source: Introduction to NumPy. Izzy Weber. Rpubs

Adding Columns

- We’ll add *trunk_stump_diameters* 1D array as a column to the *tree_census* array.



```
# Print the shapes of tree_census and trunk_stump_diameters
print(trunk_stump_diameters.shape, tree_census.shape)

# Reshape trunk_stump_diameters
reshaped_diameters = trunk_stump_diameters.reshape((1000, 1))

# Concatenate reshaped_diameters to tree_census as the last column
concatenated_tree_census = np.concatenate((tree_census, reshaped_diameters), axis=1)
print(concatenated_tree_census)
```

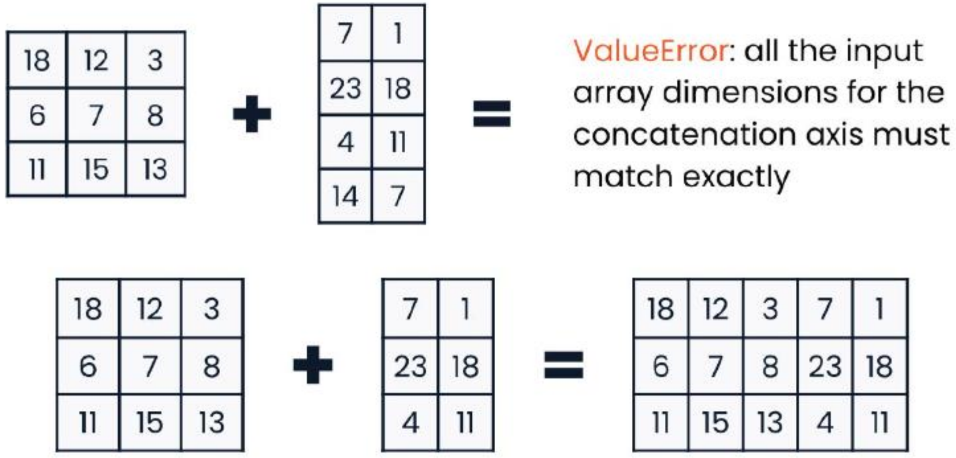


```
>>> print(trunk_stump_diameters.shape, tree_census.shape)
(1000,) (1000, 4)
>>>
>>> print(concatenated_tree_census)
[[     3 501451     24      0     24]
 [     4 501451     20      0     20]
 [     7 501911      3      0      3]
 ...
 [  1198 227387     11      0     11]
 [  1199 227387     11      0     11]
 [  1210 227386      6      0      6]]
```

Source: Introduction to NumPy. Izzy Weber. Rpubs

Compatibility for Concatenation

- Arrays must have the same shape along all axes except the one being concatenated along.



18	12	3
6	7	8
11	15	13

+

7	1
23	18
4	11
14	7

=

ValueError: all the input array dimensions for the concatenation axis must match exactly

18	12	3
6	7	8
11	15	13

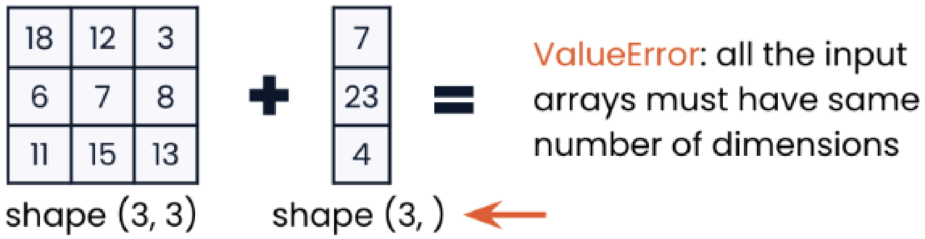
+

7	1
23	18
4	11

=

18	12	3	7	1
6	7	8	23	18
11	15	13	4	11

- Two arrays must also have the same number of dimension.



18	12	3
6	7	8
11	15	13

shape (3, 3)

+

7
23
4

shape (3,)

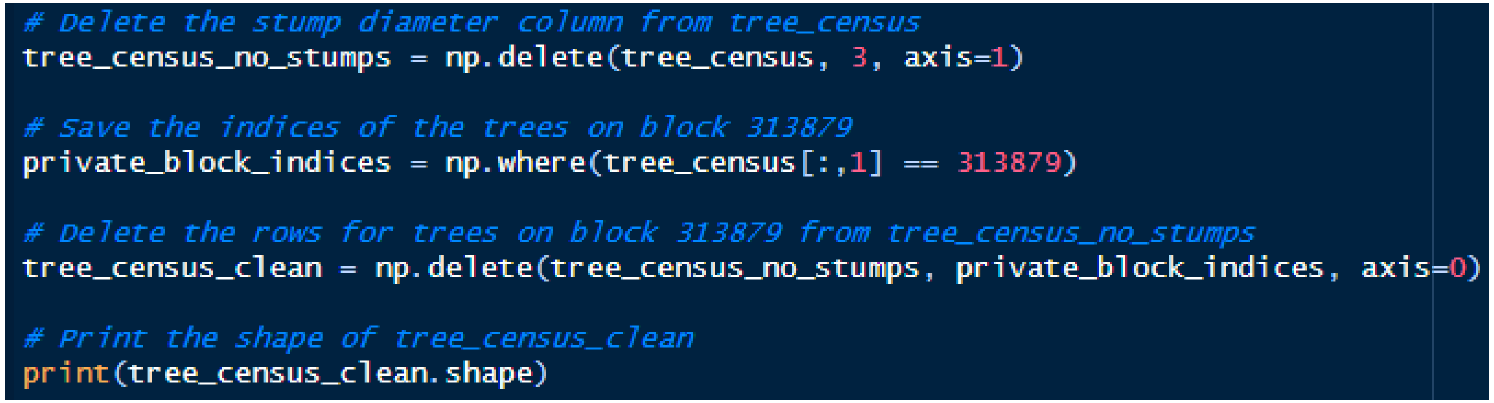
=

ValueError: all the input arrays must have same number of dimensions

Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Deleting with np.delete()

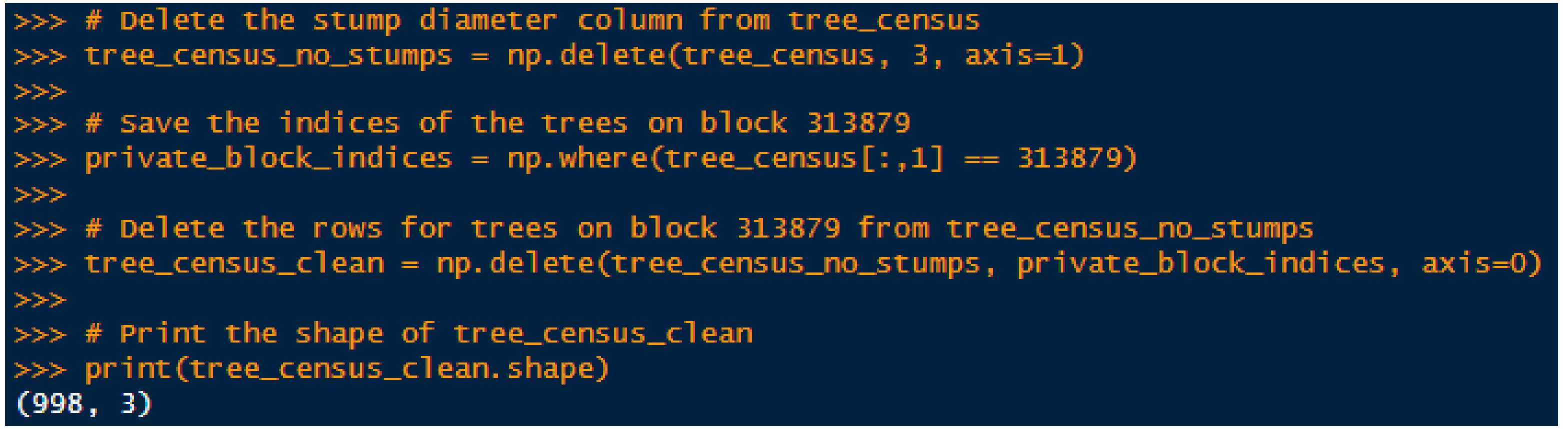
- It might be helpful to delete some unneeded data.



```python
# Delete the stump diameter column from tree_census
tree_census_no_stumps = np.delete(tree_census, 3, axis=1)

# Save the indices of the trees on block 313879
private_block_indices = np.where(tree_census[:,1] == 313879)

# Delete the rows for trees on block 313879 from tree_census_no_stumps
tree_census_clean = np.delete(tree_census_no_stumps, private_block_indices, axis=0)

# Print the shape of tree_census_clean
print(tree_census_clean.shape)
```

Explanation:

- *np.delete()* function takes three arguments: the original array, the index or indices to be deleted, and the axis to delete along. *np.where()* returns an array of indices where a condition is met



```
>>> # Delete the stump diameter column from tree_census
>>> tree_census_no_stumps = np.delete(tree_census, 3, axis=1)
>>>
>>> # Save the indices of the trees on block 313879
>>> private_block_indices = np.where(tree_census[:,1] == 313879)
>>>
>>> # Delete the rows for trees on block 313879 from tree_census_no_stumps
>>> tree_census_clean = np.delete(tree_census_no_stumps, private_block_indices, axis=0)
>>>
>>> # Print the shape of tree_census_clean
>>> print(tree_census_clean.shape)
(998, 3)
```

Source: Introduction to NumPy. Izzy Weber. Rpubs

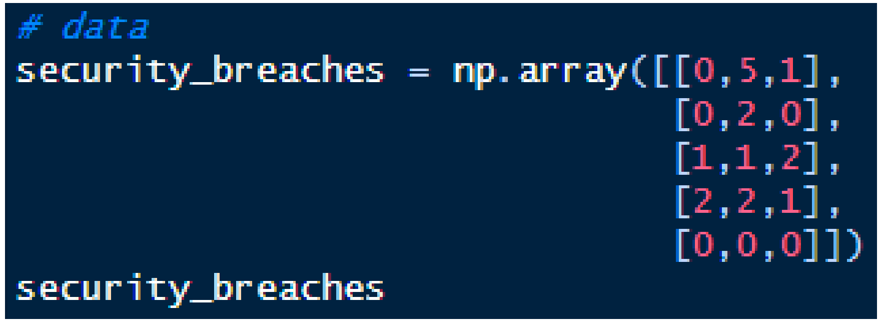
7. Array Mathematics

Summarizing Data

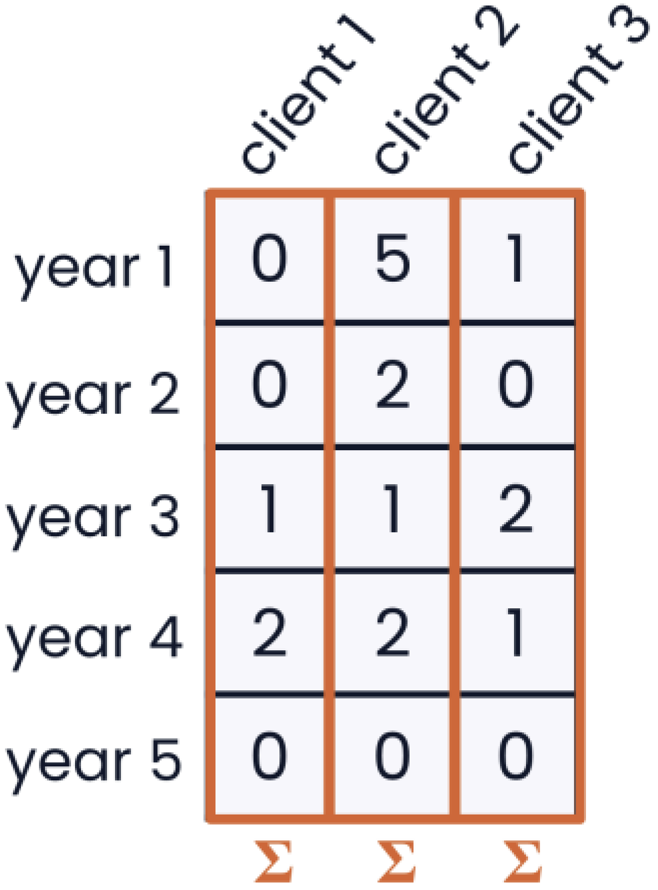
- To have an aggregate picture about our data we may use:
  - .sum()
  - .min()
  - .max()
  - .mean()
  - .cumsum()

Our data:

- *Security breaches of 3 clients per year.*



```
# data
security_breaches = np.array([[0,5,1],
                              [0,2,0],
                              [1,1,2],
                              [2,2,1],
                              [0,0,0]])
security_breaches
```

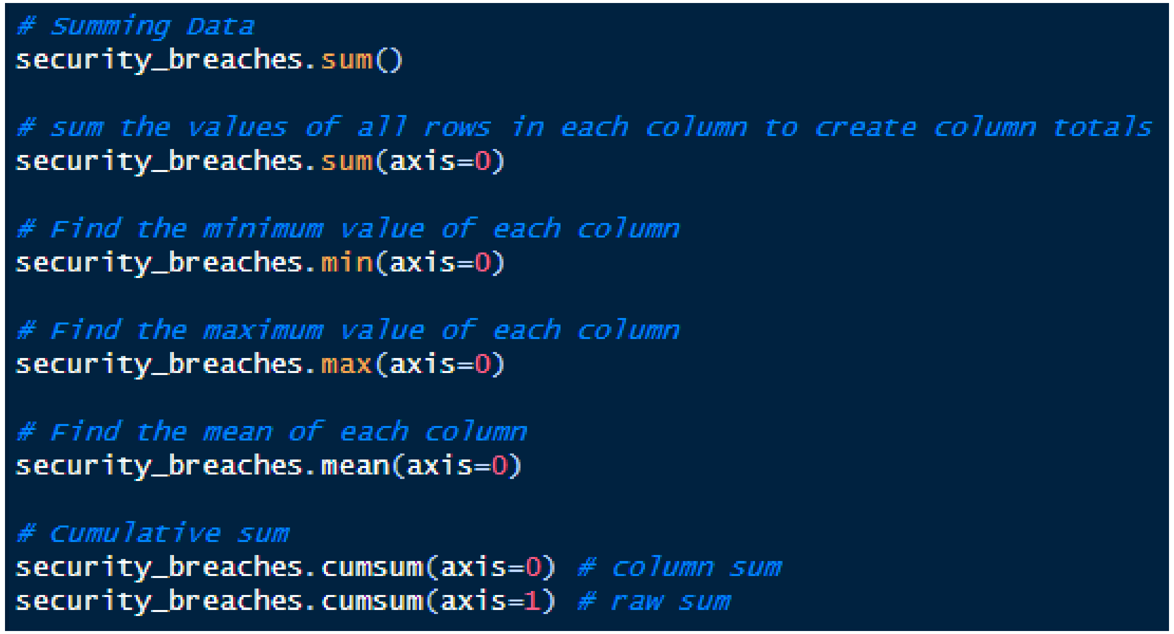


	client 1	client 2	client 3
year 1	0	5	1
year 2	0	2	0
year 3	1	1	2
year 4	2	2	1
year 5	0	0	0
	Σ	Σ	Σ

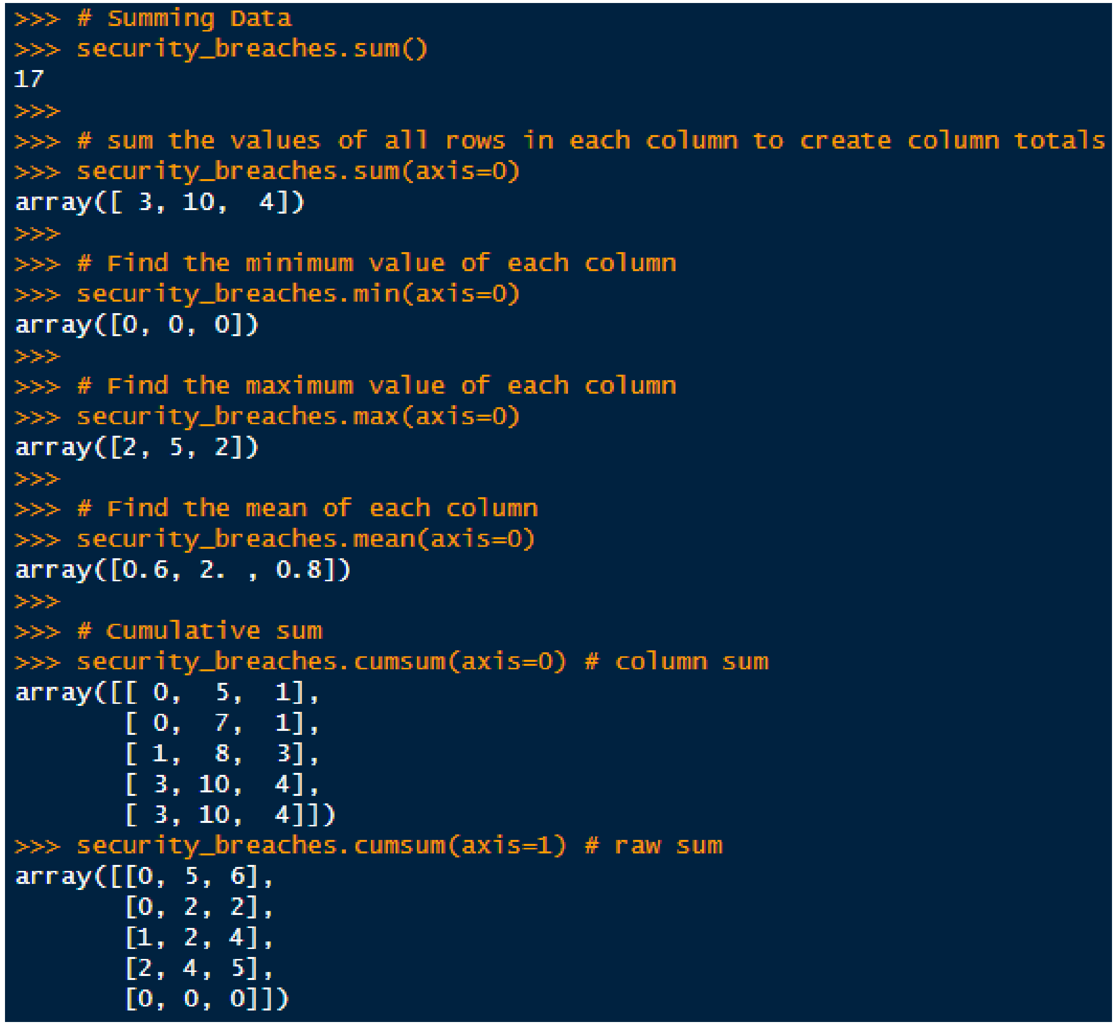
Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Summarizing Data (2)

- Let’s work with *Security breaches* data:



```
# Summing Data
security_breaches.sum()

# sum the values of all rows in each column to create column totals
security_breaches.sum(axis=0)

# Find the minimum value of each column
security_breaches.min(axis=0)

# Find the maximum value of each column
security_breaches.max(axis=0)

# Find the mean of each column
security_breaches.mean(axis=0)

# Cumulative sum
security_breaches.cumsum(axis=0) # column sum
security_breaches.cumsum(axis=1) # raw sum
```

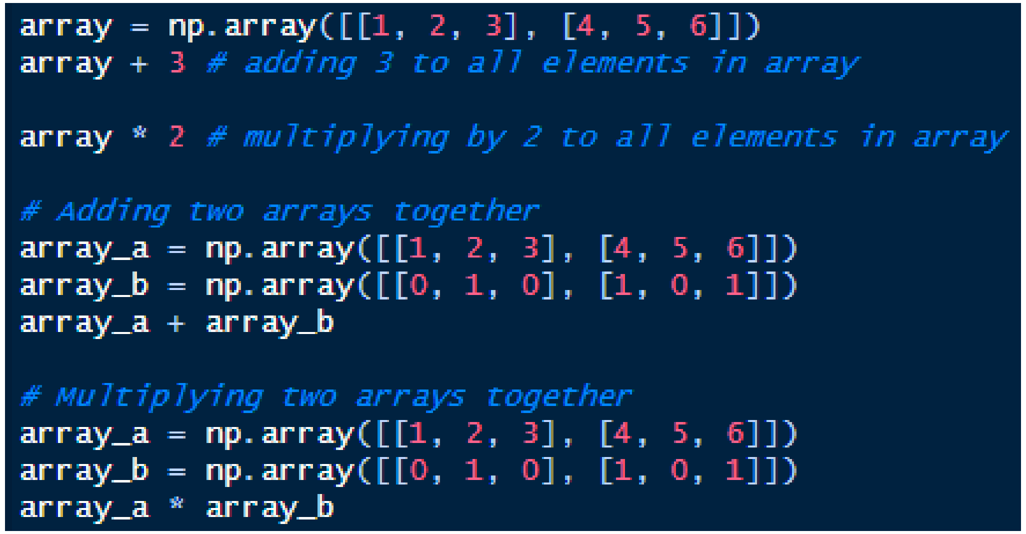


```
>>> # Summing Data
>>> security_breaches.sum()
17
>>>
>>> # sum the values of all rows in each column to create column totals
>>> security_breaches.sum(axis=0)
array([ 3, 10,  4])
>>>
>>> # Find the minimum value of each column
>>> security_breaches.min(axis=0)
array([0, 0, 0])
>>>
>>> # Find the maximum value of each column
>>> security_breaches.max(axis=0)
array([2, 5, 2])
>>>
>>> # Find the mean of each column
>>> security_breaches.mean(axis=0)
array([0.6, 2.  , 0.8])
>>>
>>> # Cumulative sum
>>> security_breaches.cumsum(axis=0) # column sum
array([[ 0,  5,  1],
       [ 0,  7,  1],
       [ 1,  8,  3],
       [ 3, 10,  4],
       [ 3, 10,  4]])
>>> security_breaches.cumsum(axis=1) # raw sum
array([[0, 5, 6],
       [0, 2, 2],
       [1, 2, 4],
       [2, 4, 5],
       [0, 0, 0]])
```

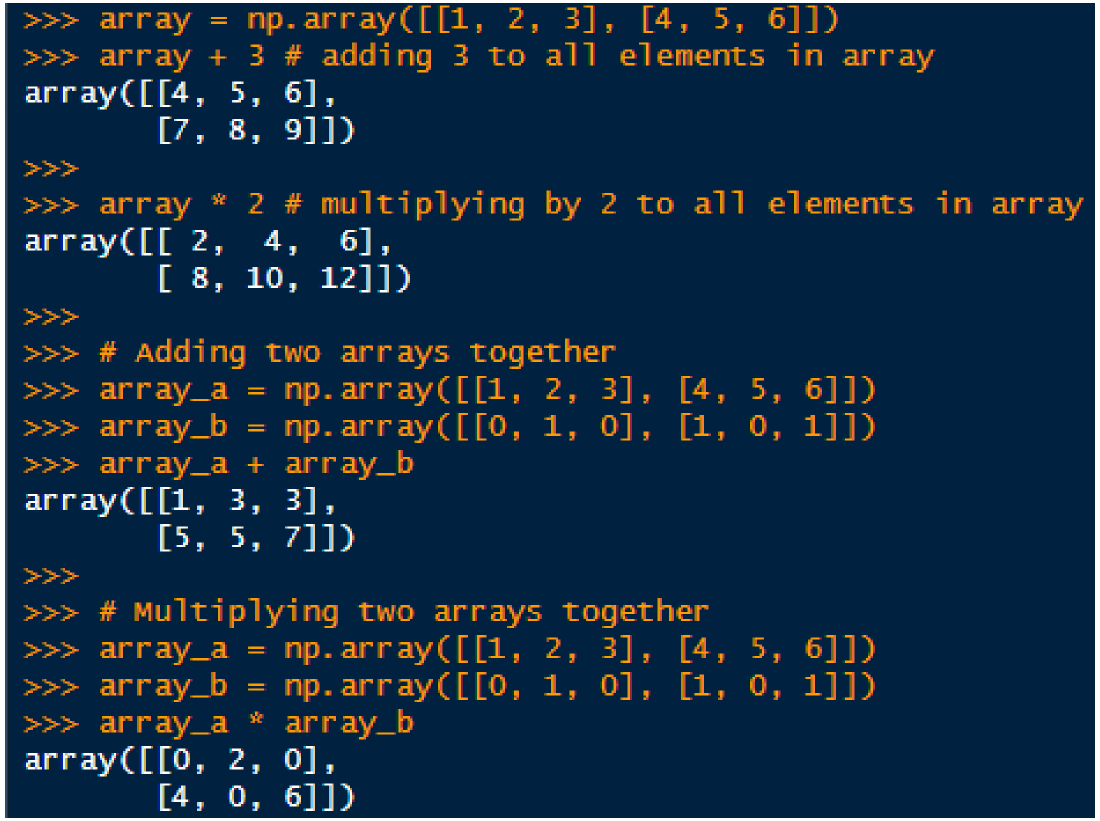
Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Vectorized operations

- Extension of the laws of elementary algebra to vectors. They include addition, subtraction, and three types of multiplication.



```
array = np.array([[1, 2, 3], [4, 5, 6]])
array + 3 # adding 3 to all elements in array

array * 2 # multiplying by 2 to all elements in array

# Adding two arrays together
array_a = np.array([[1, 2, 3], [4, 5, 6]])
array_b = np.array([[0, 1, 0], [1, 0, 1]])
array_a + array_b

# Multiplying two arrays together
array_a = np.array([[1, 2, 3], [4, 5, 6]])
array_b = np.array([[0, 1, 0], [1, 0, 1]])
array_a * array_b
```

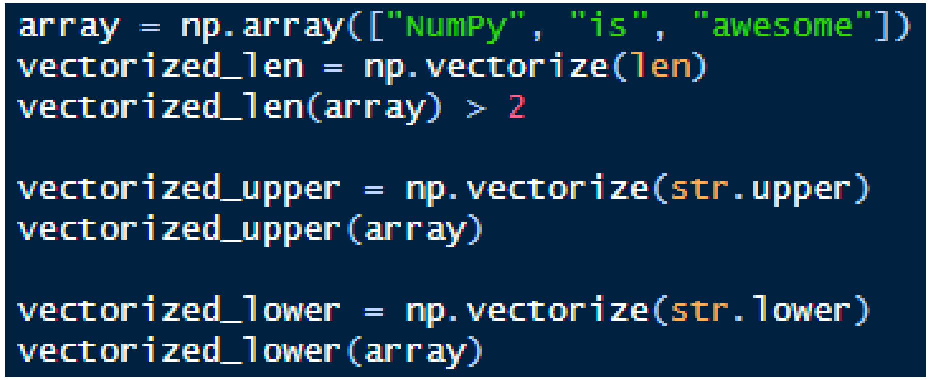


```
>>> array = np.array([[1, 2, 3], [4, 5, 6]])
>>> array + 3 # adding 3 to all elements in array
array([[4, 5, 6],
       [7, 8, 9]])
>>>
>>> array * 2 # multiplying by 2 to all elements in array
array([[ 2,  4,  6],
       [ 8, 10, 12]])
>>>
>>> # Adding two arrays together
>>> array_a = np.array([[1, 2, 3], [4, 5, 6]])
>>> array_b = np.array([[0, 1, 0], [1, 0, 1]])
>>> array_a + array_b
array([[1, 3, 3],
       [5, 5, 7]])
>>>
>>> # Multiplying two arrays together
>>> array_a = np.array([[1, 2, 3], [4, 5, 6]])
>>> array_b = np.array([[0, 1, 0], [1, 0, 1]])
>>> array_a * array_b
array([[0, 2, 0],
       [4, 0, 6]])
```

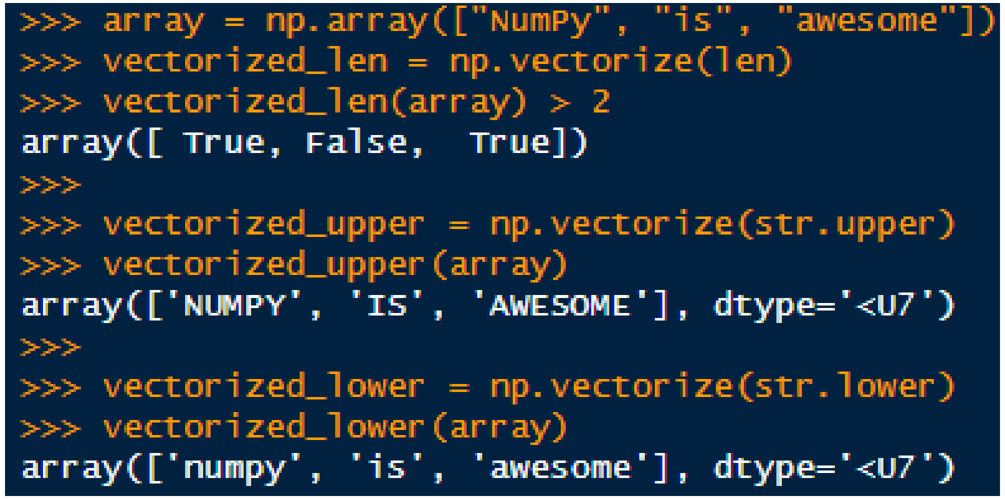
Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Vectorized operations (2)

- *np.vectorize()* is used to vectorize Python methods and functions on array elements in NumPy. Vectorized operations are much faster and more efficient than a for loop
- Make all the letters of every word in the array uppercase/lowercase by using *np.vectorize()*



```
array = np.array(["NumPy", "is", "awesome"])
vectorized_len = np.vectorize(len)
vectorized_len(array) > 2

vectorized_upper = np.vectorize(str.upper)
vectorized_upper(array)

vectorized_lower = np.vectorize(str.lower)
vectorized_lower(array)
```



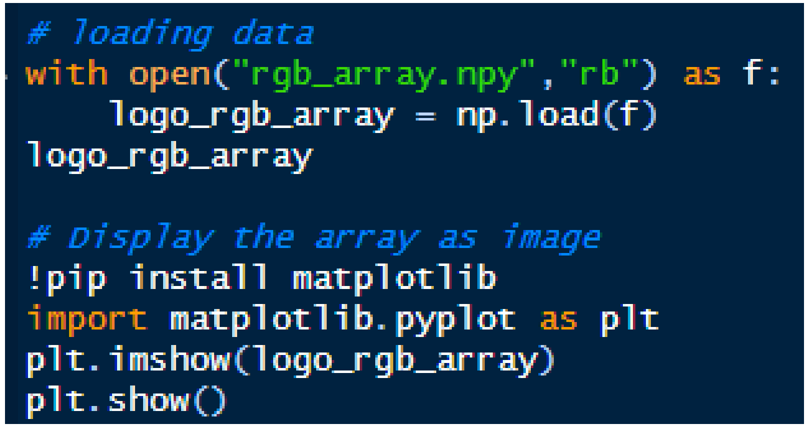
```
>>> array = np.array(["NumPy", "is", "awesome"])
>>> vectorized_len = np.vectorize(len)
>>> vectorized_len(array) > 2
array([ True, False,  True])
>>>
>>> vectorized_upper = np.vectorize(str.upper)
>>> vectorized_upper(array)
array(['NUMPY', 'IS', 'AWESOME'], dtype='<U7')
>>>
>>> vectorized_lower = np.vectorize(str.lower)
>>> vectorized_lower(array)
array(['numpy', 'is', 'awesome'], dtype='<U7')
```

Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

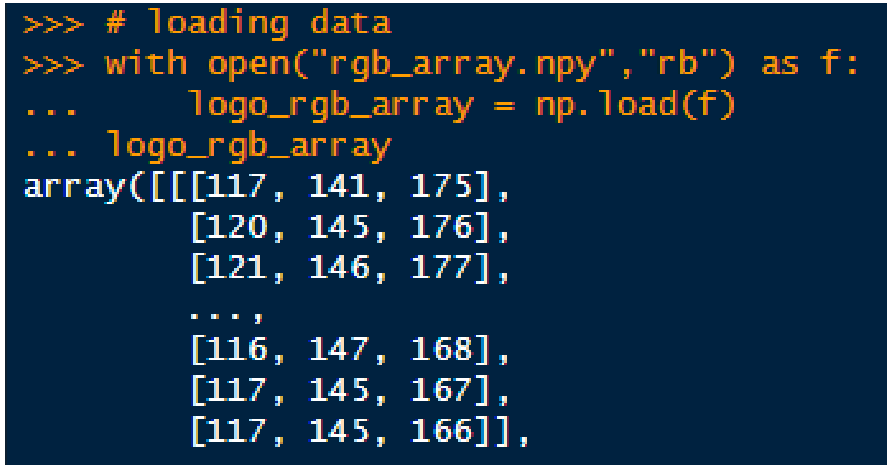
8. Array Transformations

Loading Arrays

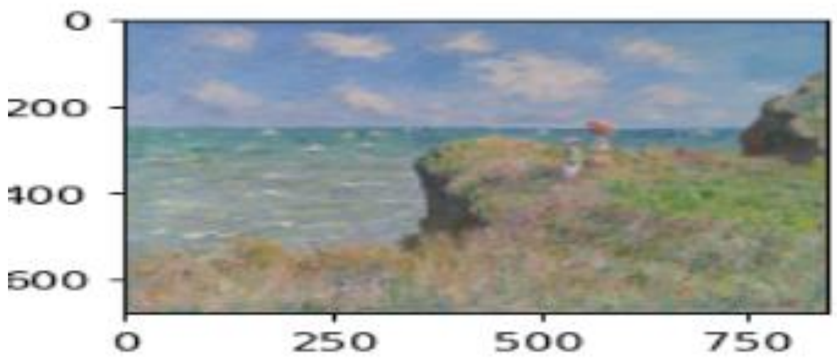
- *Let’s* load *rgb_array.npy* from the folder



```
# loading data
with open("rgb_array.npy","rb") as f:
    logo_rgb_array = np.load(f)
logo_rgb_array

# Display the array as image
!pip install matplotlib
import matplotlib.pyplot as plt
plt.imshow(logo_rgb_array)
plt.show()
```



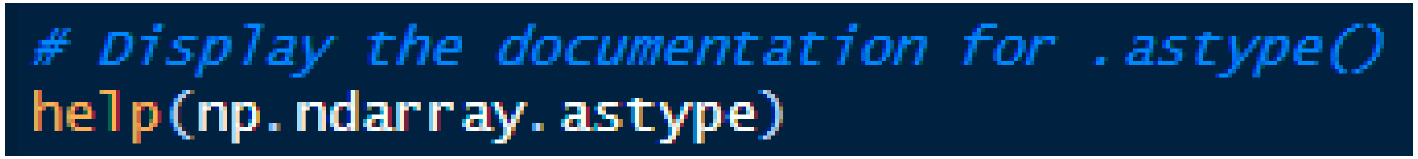
```
>>> # loading data
>>> with open("rgb_array.npy","rb") as f:
...     logo_rgb_array = np.load(f)
... logo_rgb_array
array([[[117, 141, 175],
        [120, 145, 176],
        [121, 146, 177],
        ...,
        [116, 147, 168],
        [117, 145, 167],
        [117, 145, 166]],
```



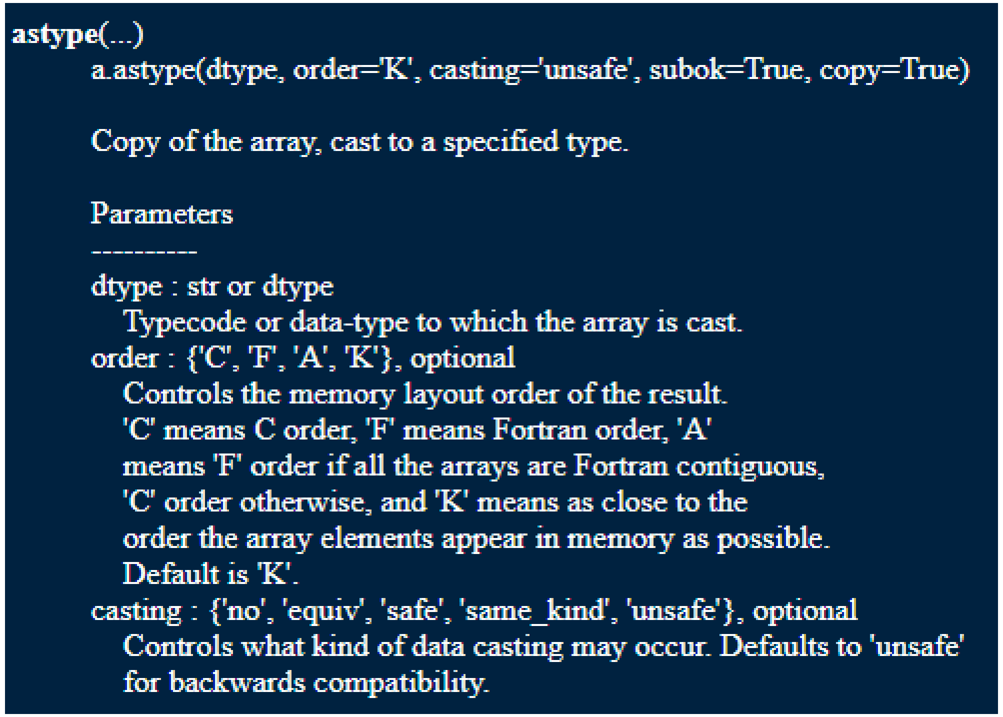
Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Getting help

- Return NumPy’s documentation text for *.astype()*



```
# Display the documentation for .astype()
help(np.ndarray.astype)
```

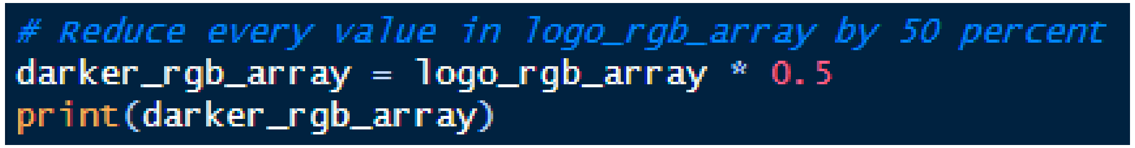


```
astype(...)
    a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)

    Copy of the array, cast to a specified type.

    Parameters
    ----------
    dtype : str or dtype
        Typecode or data-type to which the array is cast.
    order : {'C', 'F', 'A', 'K'}, optional
        Controls the memory layout order of the result.
        'C' means C order, 'F' means Fortran order, 'A'
        means 'F' order if all the arrays are Fortran contiguous,
        'C' order otherwise, and 'K' means as close to the
        order the array elements appear in memory as possible.
        Default is 'K'.
    casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
        Controls what kind of data casting may occur. Defaults to 'unsafe'
        for backwards compatibility.
```

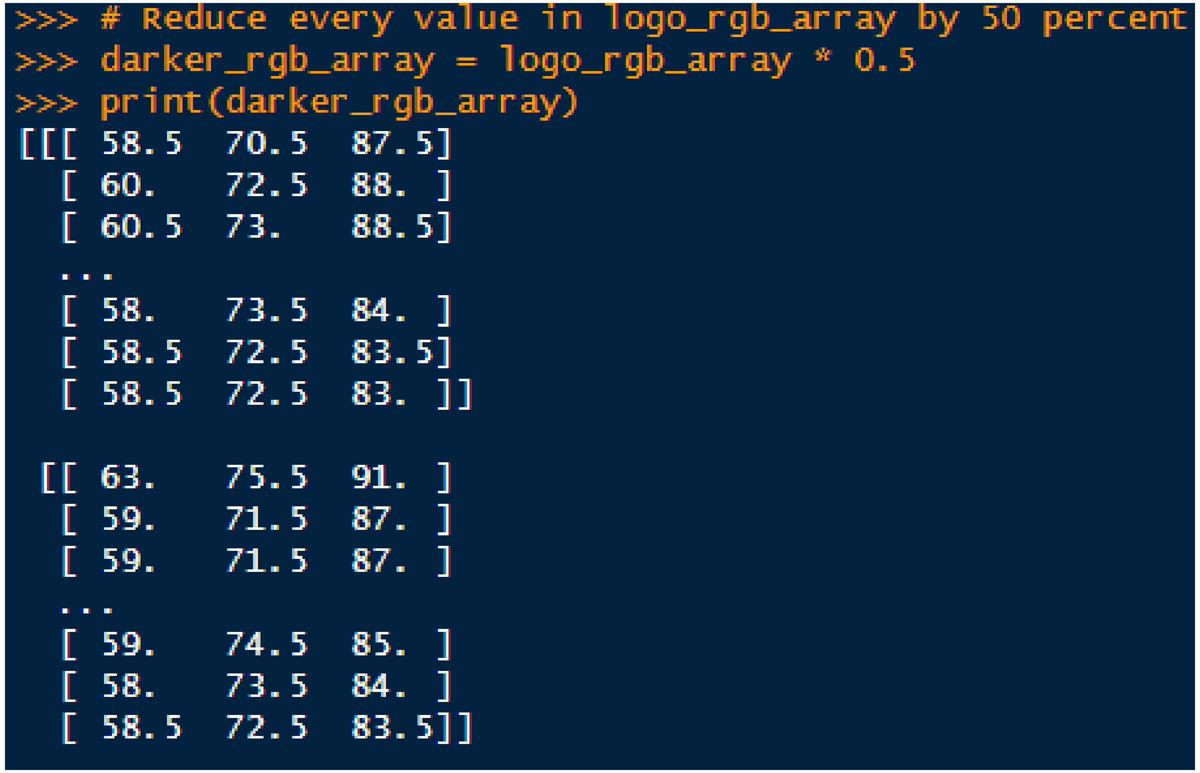
Source: Introduction to NumPy. Izzy Weber. Rpubs

Updating Array

- *Let’s* update *logo_rgb_array*



```
# Reduce every value in logo_rgb_array by 50 percent
darker_rgb_array = logo_rgb_array * 0.5
print(darker_rgb_array)
```

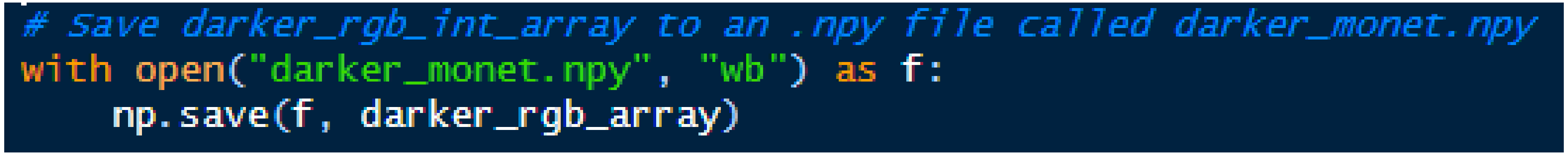


```
>>> # Reduce every value in logo_rgb_array by 50 percent
>>> darker_rgb_array = logo_rgb_array * 0.5
>>> print(darker_rgb_array)
[[[ 58.5  70.5  87.5]
  [ 60.   72.5  88. ]
  [ 60.5  73.   88.5]
  ...
  [ 58.   73.5  84. ]
  [ 58.5  72.5  83.5]
  [ 58.5  72.5  83. ]]

 [[ 63.   75.5  91. ]
  [ 59.   71.5  87. ]
  [ 59.   71.5  87. ]
  ...
  [ 59.   74.5  85. ]
  [ 58.   73.5  84. ]
  [ 58.5  72.5  83.5]]
```

Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Saving Array

- *Let's* save *darker_rgb_array*



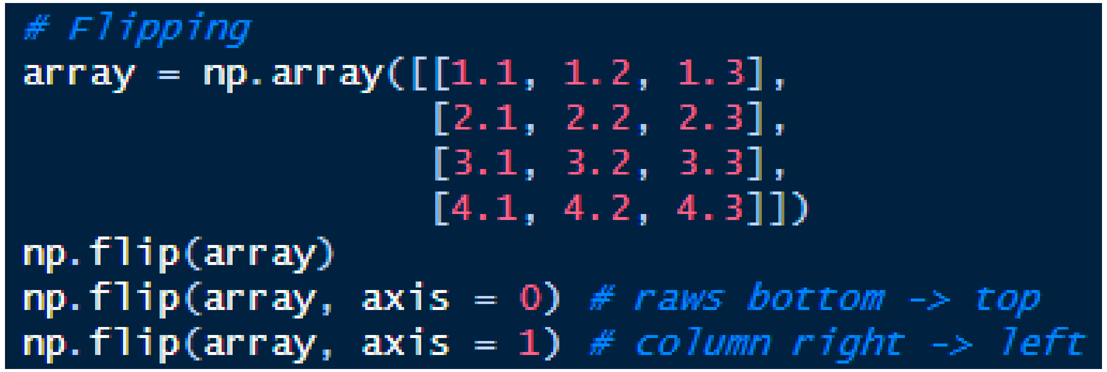
```
# Save darker_rgb_int_array to an .npy file called darker_monet.npy
with open("darker_monet.npy", "wb") as f:
    np.save(f, darker_rgb_array)
```

- Check your folder for the new file in there.

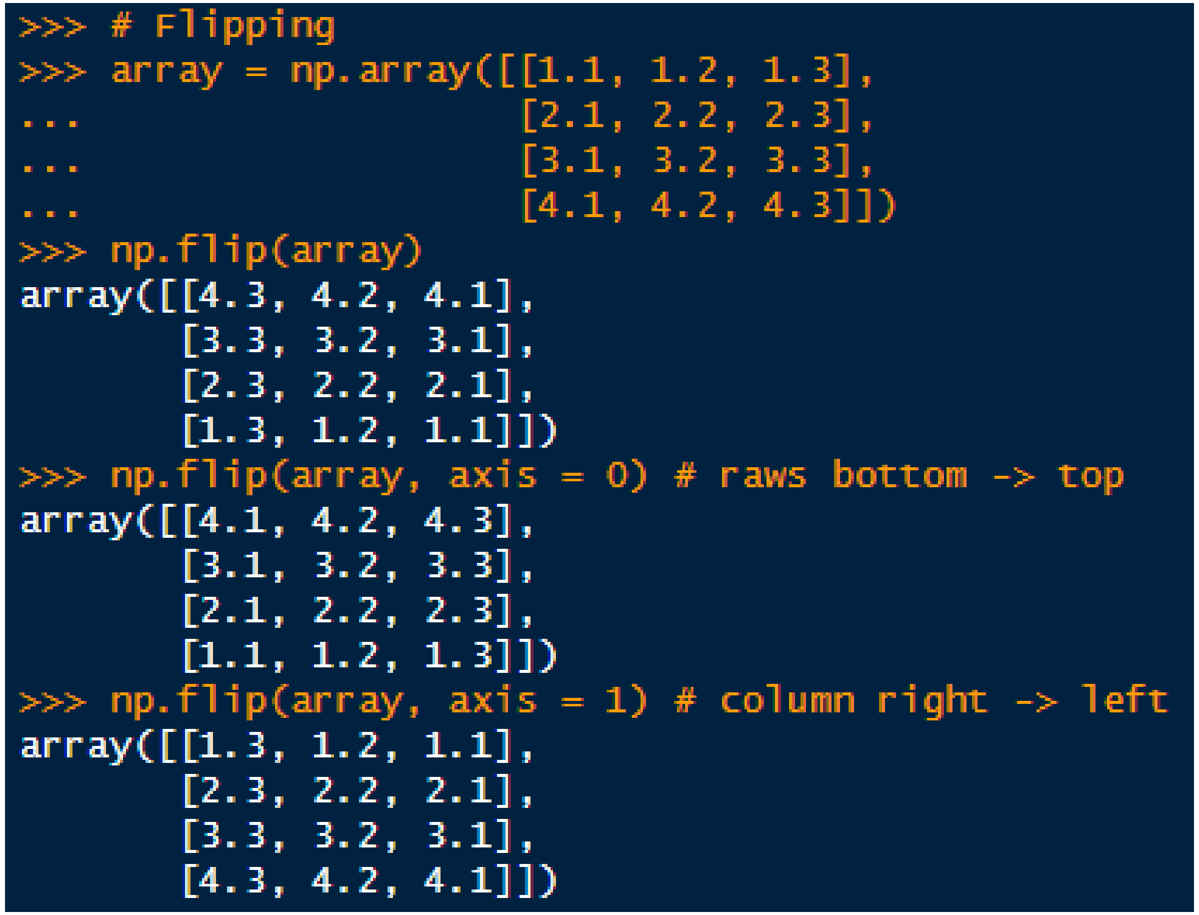
Source: Introduction to NumPy. Izzy Weber. Rpubs

Flipping Array

- Reverse the order of elements in an array along the given axis.



```
# Flipping
array = np.array([[1.1, 1.2, 1.3],
                  [2.1, 2.2, 2.3],
                  [3.1, 3.2, 3.3],
                  [4.1, 4.2, 4.3]])
np.flip(array)
np.flip(array, axis = 0) # raws bottom -> top
np.flip(array, axis = 1) # column right -> left
```

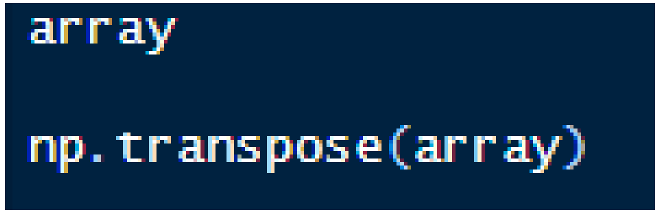


```
>>> # Flipping
>>> array = np.array([[1.1, 1.2, 1.3],
...                   [2.1, 2.2, 2.3],
...                   [3.1, 3.2, 3.3],
...                   [4.1, 4.2, 4.3]])
>>> np.flip(array)
array([[4.3, 4.2, 4.1],
       [3.3, 3.2, 3.1],
       [2.3, 2.2, 2.1],
       [1.3, 1.2, 1.1]])
>>> np.flip(array, axis = 0) # raws bottom -> top
array([[4.1, 4.2, 4.3],
       [3.1, 3.2, 3.3],
       [2.1, 2.2, 2.3],
       [1.1, 1.2, 1.3]])
>>> np.flip(array, axis = 1) # column right -> left
array([[1.3, 1.2, 1.1],
       [2.3, 2.2, 2.1],
       [3.3, 3.2, 3.1],
       [4.3, 4.2, 4.1]])
```

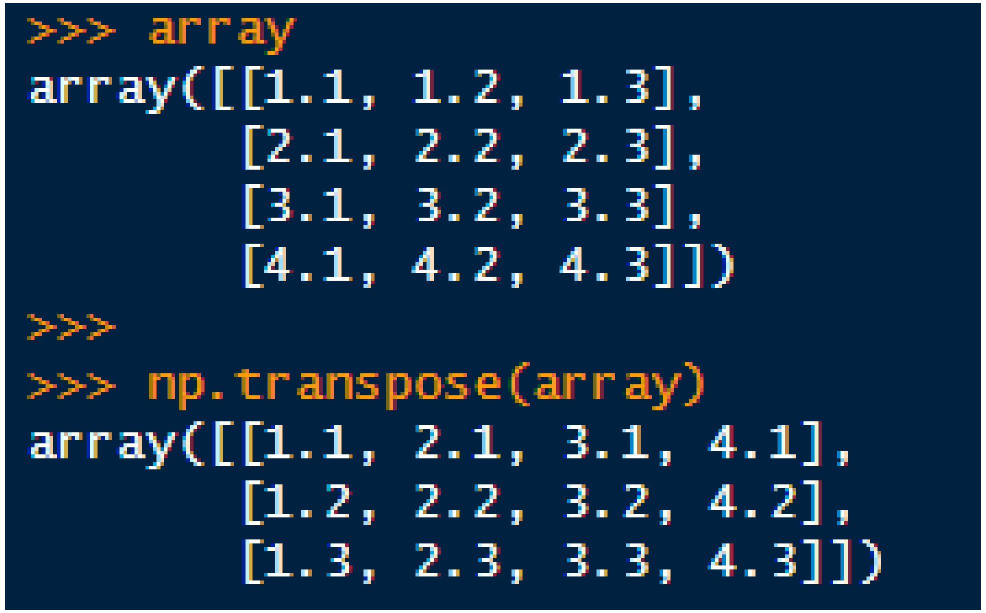
Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

Transposing Array

- This function permutes or reserves the dimension of the given array and returns the modified array.



```
array

np.transpose(array)
```



```
>>> array
array([[1.1, 1.2, 1.3],
       [2.1, 2.2, 2.3],
       [3.1, 3.2, 3.3],
       [4.1, 4.2, 4.3]])
>>>
>>> np.transpose(array)
array([[1.1, 2.1, 3.1, 4.1],
       [1.2, 2.2, 3.2, 4.2],
       [1.3, 2.3, 3.3, 4.3]])
```

Source: Kaggle. Intro to Numpy. https://www.kaggle.com/code/sharifulislams/introduction-to-numpy/notebook

9. In-class Assignment

Instructions

Please open the DataCamp Group and do the following:

- Complete Chapters 3 & 4 of the Introduction to Python course.
- *For students with Intermediate Python* – Please do Data-Driven Product Management: Conducting a Market Analysis.
- Please don’t use the DataCamp in-build AI helper.

- Submit the screenshot showing the completion of these chapters and every assignment there.

It's an individual assignment.

Max score: 10 points – for two chapters, 7 points for one chapter.

Q & A

Thank you!