Python For Data Analysis

A high-level, open-source, general programming language

Outline

1. Intro to Data Science
2. IPython
   - Jupyter Notebook
3. Numpy
   - Arrays vs Lists
   - Working with Arrays

Prerequisites

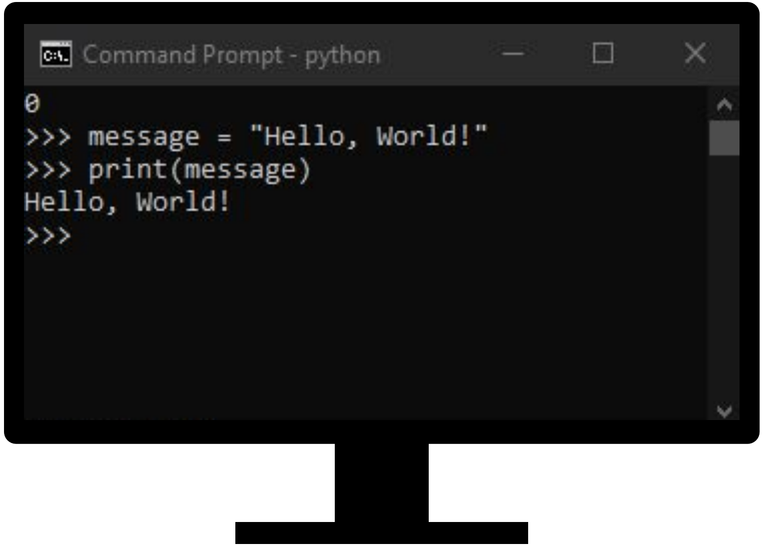
1. https://www.python.org/downloads/
   - Download Python for your Operating System
2. https://code.visualstudio.com/
   - Visual Studio Code is the current standard for Integrated Development Environments
   - The **Python** and **Pylance** extensions are recommended
3. https://www.anaconda.com/products/individual
   - Data focused distribution
   - The Anaconda distribution provides a suite of tools for data science

IP[y]:

IPython

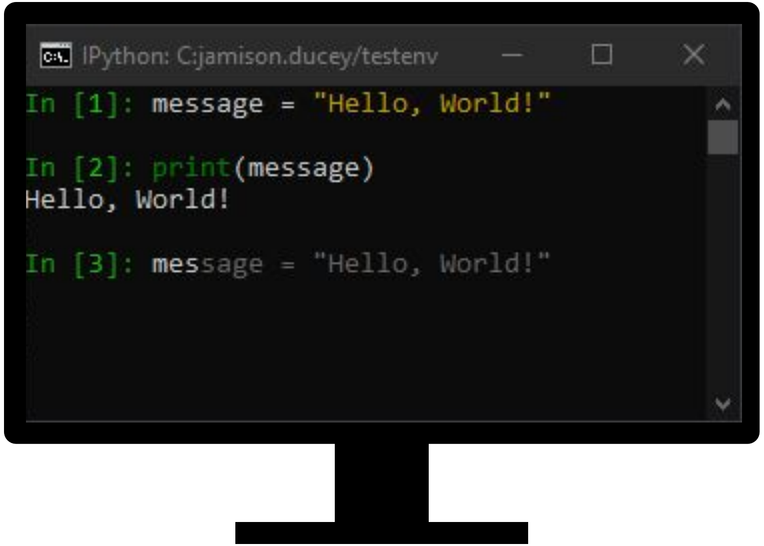
The Python Interpreter

- Python's interpreter is **interactive**
  - REPL is our primary mode of utilizing Python
    - **Read**, **Execute**, **Print**, **Loop**
  - Alternative to REPL - running files
- Some characteristics of REPLs
  - State is **ephemeral**
  - History is **immutable**
  - Paradigm is **declarative**



The IPython Interpreter

- Built off of Python
  - Still using REPL, but ~**enhanced**~
    - Syntax Highlighting
    - Code Completion
    - Kernel for **Jupyter**
    - And much, much more!



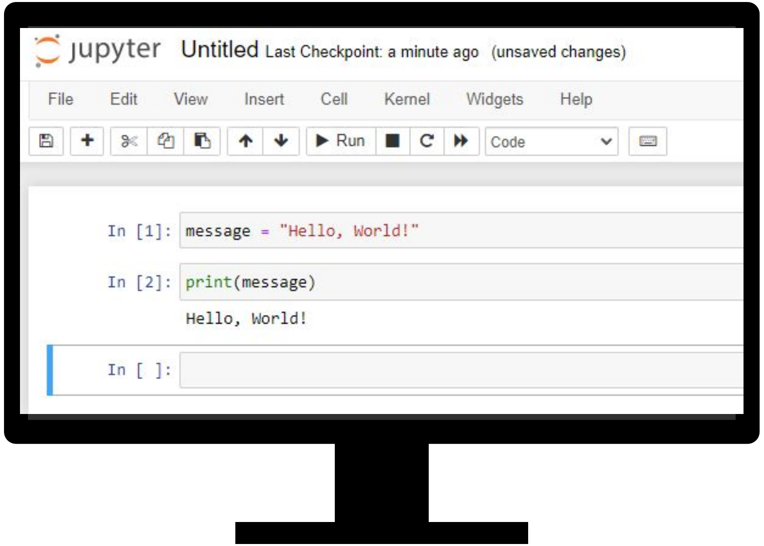
Jupyter Notebooks

Notebooks vs REPL

	state	history	language
Notebook	persistent	modifiable	imperative
REPL	ephemeral	immutable	declarative

IPython via Jupyter

- ➔ Notebooks are the **document** version of REPLs
  - ◆ **Individually runnable** code cells
  - ◆ Cells can be run in **any order** - important to keep track of this!
  - ◆ Easy viewing/modification of **local history** (as opposed to REPLs)
  - ◆ EXCELLENT for data - **imperative** paradigm



Numpy

Numpy

- **Numpy** is a python library for working with ***arrays***
  - Numpy supports arrays of multiple dimensions
- Python does not **natively** support arrays
- Arrays are **faster** than lists
  - Contiguous in memory
- Numpy is primarily written in **c++**
  - Created in 2005
  - Open Source

Import Numpy

- ➔ Numpy isn't a built-in module
  - pip install numpy
  - conda install numpy
- ➔ Numpy is usually imported with the alias np - easier to read
- ➔ __version__ will return the numpy version

```python
import numpy as np

ver =  np.__version__

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

type(arr) # numpy.ndarray
```

Python Arrays vs. Numpy Lists

- ➔ Python Lists are easy to work with, and have fewer restrictions
- ➔ Numpy Lists are faster and more memory efficient with large data sets

Python Lists	Numpy Arrays
Elements are treated as objects	Elements are contiguous in memory
Can store any data types at once	All elements must be the same data type
Lists can be altered	Arrays are recreated internally if an element is changed

Array Basics

→ Array: A grid of values that can be indexed in various ways, all of the same type, called **dtype**

→ An array's **rank** is its number of dimensions, while its **shape** is a tuple of integers giving the size of the array along each dimension

→ Elements can be accessed in the same manner as lists - via square brackets

Rank: 1, Shape: (6)

```
a = np.array([2, 4, 6, 8, 10, 12])
```

Rank: 2, Shape: (3, 2)

```
b = np.array([[2, 4], [6, 8], [10, 12]])
```

Evaluates to True

```
a[3] == b[1][1]
```

More on Arrays

- ➔ **ndarrays** = "N-dimensional" arrays
  - If N = 1: array is 1-dimensional / 1-D
  - If N = 2: array is 2-dimensional / 2-D
  - etc.
- ➔ **Vectors** are 1-D arrays
- ➔ **Matrices** (s. Matrix) are 2-D arrays
- ➔ **Tensors** are 3(+)-D arrays
- ➔ In NumPy, dimensions are referred to as **axes**.
- ➔ Different arrays can share the **same** data - changes made to one might be visible in another.
- ➔ **Attributes** = information intrinsic to the array itself

2nd Axis, Length of

1st Axis, Length of 2

```
[[0., 0., 0.],
 [1., 1., 1.]]
```

Creating Arrays

➔ Many different methods for this:

- **.array(list)** - pass a list to turn it into an array
- **.zeros(shape)** / **.ones(shape)** / **.empty(shape)** - pass a shape to fill the array with that particular number (empty is random - fastest)
- **.arange([start,]stop, [step,])** - use a range to build the array
- **.linspace(start, stop, num)** - linear values along a specified interval

```
a1 = np.array([1, 2, 3])                 [1 2 3]
a2 = np.zeros(2)                          [0. 0.]
a3 = np.ones(2)                           [1. 1.]
a4 = np.empty(3)     [0.0e+0 1.3e-3 0.0e+0]
a5 = np.arange(3)                         [0 1 2]
a6 = np.arange(2, 9, 2)                 [2 4 6 8]
a7 = np.linspace(0, 5, num=3)          [0 2.5 5]
```

Data Type

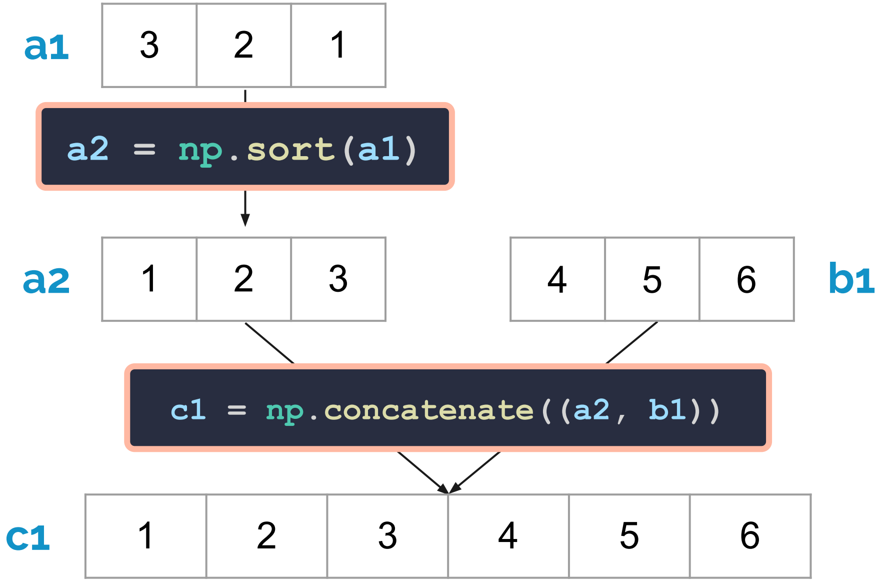
- **Arrays** must be of a single **dtype**
- Python is a **dynamically** typed language, but it is still strongly typed
- **Numpy** has its own internal set of types for arrays
- When an array is created, all elements are cast to the most **general type** in the array definition
- An array can be **recast**, but only to a compatible type
- A dtype **argument** is available for all array-creation functions covered in the previous slide.
  - Example - `np.ones(3, dtype=int8)`

Numpy Data Types

typestr	Data Type Name
i	integer
b	boolean
u	unsigned int
f	float
c	complex
m	timedelta
M	datetime
O	Object
S	String
U	Unicode String
V	Void

Element Operations

- Sorting
  - np.sort(arr) - sorts numbers in ascending order. Options for **axis**, **kind** (sorting algorithm), and **order** (to specify a field)
  - Other sorting functions -
    - argsort
    - lexsort
    - searchsorted
    - Partition
- Adding
  - np.concatenate(tuple of arrays) - adds arrays together
- Remove via **indexing**



Array Structure

➔ ndarray object attributes for analyzing **structure** of an array:

- ndarray.ndim - number of **axes** in array
- ndarray.shape - tuple of integers describing number of elements **along** each axis
- ndarray.size - number of elements in array (**product of elements** in array shape)

2
axes / dimensions

4 elements along axis

3 elements along axis

1	2	3	4
5	6	7	8
9	10	11	12

3*4 =
12

.ndim = 2

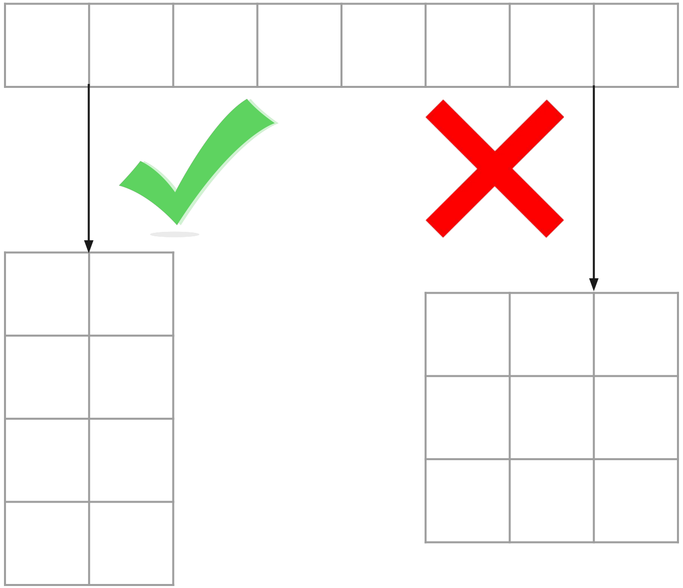
.shape = (3, 4)

.size = 12

Reshaping Arrays

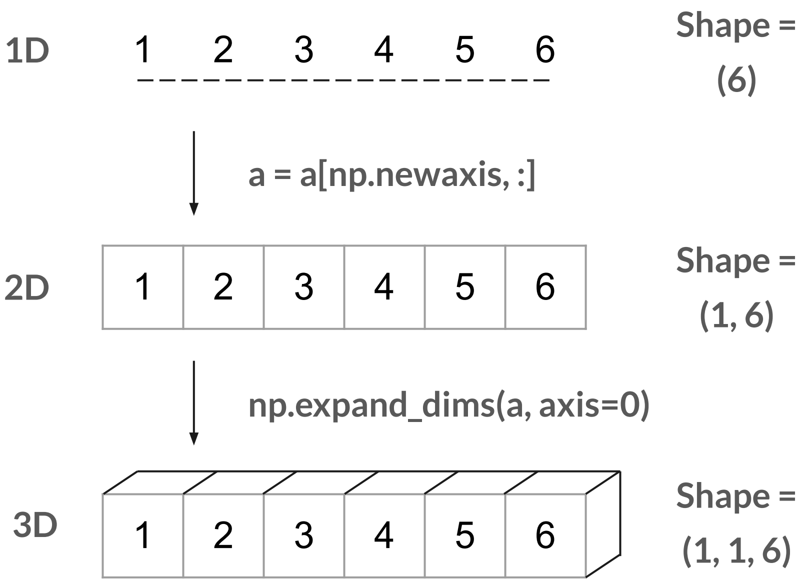
➔ ndarray.reshape(newshape)

- newshape is valid as long as its size is the **same** as the original array's size
- (8) = (4, 2)
- (8) =/= (3, 3)
- Can also use function np.reshape() to alter the original array in-place



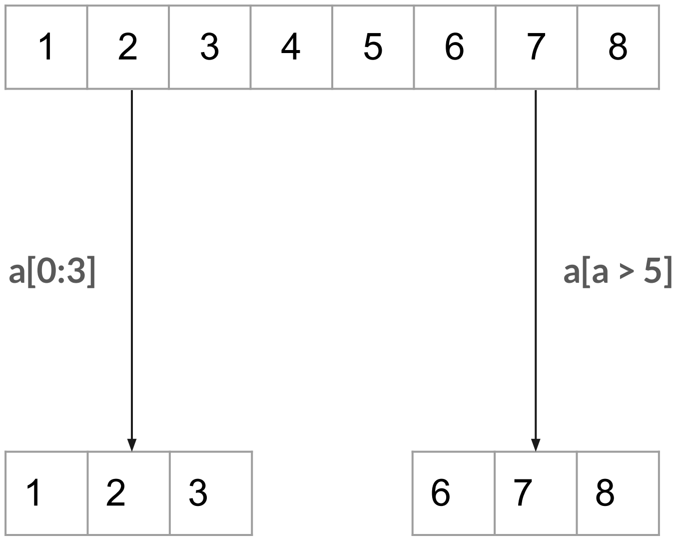
Adding Axes

- ➔ np.newaxis - increase dimensions by 1
  - 1D → 2D
  - 3D → 4D
- ➔ np.expand_dims(array, axis) - add an axis at a particular index position



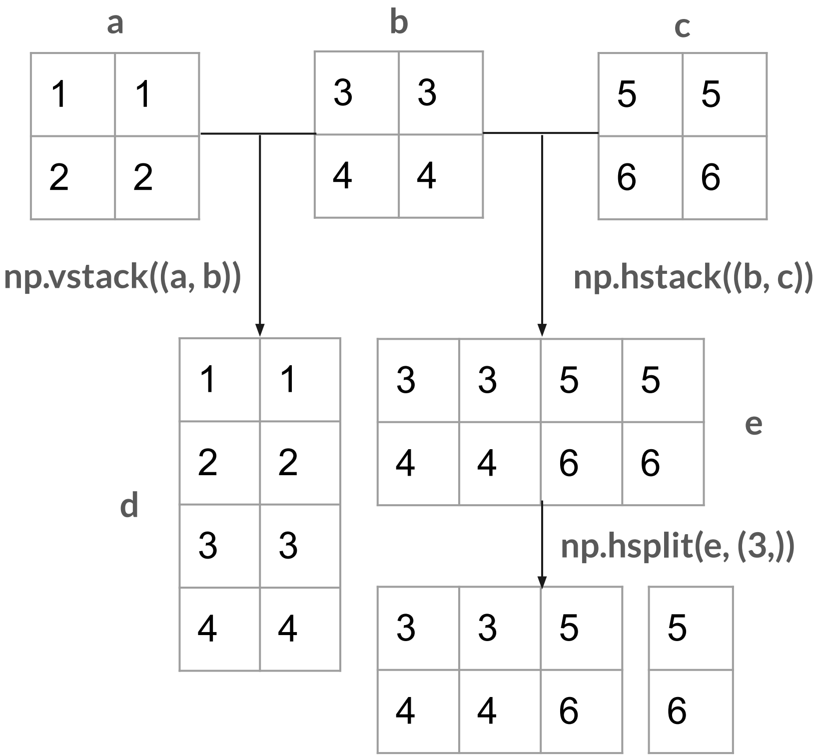
Indexing and Slicing

- Same as **lists** in Python
  - [start : stop]
- Fulfilling **conditions**:
  - arr[condition]
  - Assign conditions to **variables**
  - Can use **all comparison operators**, as well as &, |, and ^ (and, or, and xor)
- Getting **coordinates**:
  - np.nonzero(condition)
    - Generates coordinates list of matching indexes as arrays (each array is a dimension)
    - Zip result and cast to list
    - Alternatively, use result to reference elements directly



Stacking and Splitting

- Stacking - **combining** arrays
  - Vertically with np.vstack()
  - Horizontally with np.hstack()
- Split with np.hsplit(arr, sections_or_indices)
  - Always splits along axis=1
  - 2nd argument can be number of **sections** (int) or **columns** at which to split (tuple of ints)



Views and Copies

- Views are **shallow** copies
  - Returned by default whenever possible
  - Modifying data in a view modifies data in original array
  - Saves memory
- Copies are **deep** copies
  - Complete copy that can be altered with changing original array
  - .copy()

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11,
12]])
b1 = a[0, :]
b1
array([1, 2, 3, 4])
b1[0] = 99
b1
array([99,  2,  3,  4])
a
array([[99,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Array Operations

- Addition, subtraction, multiplication, and division are all available thru the usual operators
- ndarray.sum(axis)
  - **Axis** is an optional argument, if added then the function will sum **over** that axis

```
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
data + ones
array([2, 3])
data - ones
array([0, 1])
data * data
array([1, 4])
data / data
array([1., 1.])
```

Broadcasting

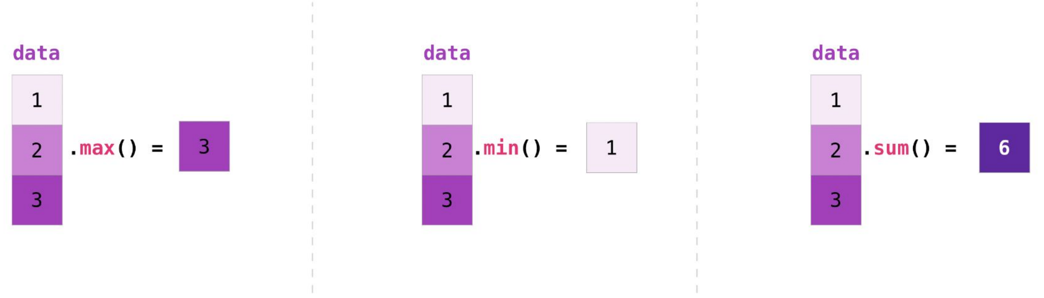
- ➔ Operating between a **vector** (array) and a **scalar** (single number) or between 2 arrays of different sizes
- ➔ Performs operations on each cell
  - Again, use the classic operators
  - +, -, *, /

```
data = np.array([1.0, 2.0])
data * 1.6
array([1.6, 3.2])
```

Array Operations cont.

More aggregation functions like .sum() are also available -

- ➔ .max() and .min()
- ➔ .mean()
- ➔ .prod() - multiplies all elements together
- ➔ .std() - standard deviation



https://numpy.org/doc/stable/_images/np_aggregation.png

RNG in NumPy

- Very useful in machine learning -
  - Generator object imported from numpy.random submodule
  - Integer generator can be made by **instantiating** generator object then calling .integers method
    - .integers(low, high, size)

```
from numpy.random import default_rng
rng = default_rng()

rng.integers(5, size=(2, 4))
array([[2, 1, 1, 0],
       [0, 0, 0, 4]])  # may vary
```

Unique Items and Counts

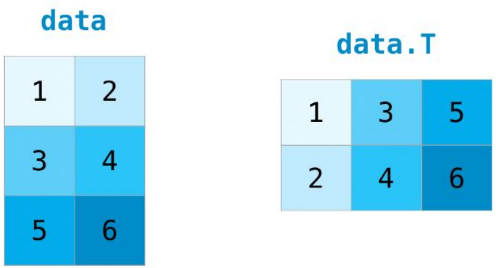
np.unique(arr) -

- Returns all **unique** values in array
- Set optional return_index argument to True -
  - Returns **indices** of unique values
- Set optional return_counts argument to True -
  - Returns **counts** of unique values
- Also works with **matrices**
  - Flattens array by default
  - For unique rows, axis = 0
  - For unique columns, axis = 1

```
a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14,
18, 19, 20])

unique_values = np.unique(a)
print(unique_values)
[11 12 13 14 15 16 17 18 19 20]

unique_values, indices_list = np.unique(a, return_index=True)
print(indices_list)
[ 0  2  3  4  5  6  7 12 13 14]

unique_values, occurrence_count = np.unique(a,
return_counts=True)
print(occurrence_count)
[3 2 2 2 1 1 1 1 1 1]
```

Transposing Matrices

- ndarray.T property -
  - Flipped version of original matrix
  - Can also use .transpose()
  - .reshape() is often used alongside .transpose()



https://numpy.org/doc/stable/_images/np_transposing_reshaping.png

```
arr = np.arange(6).reshape((2, 3))
arr
array([[0, 1, 2],
       [3, 4, 5]])
arr.transpose() # or .T
array([[0, 3],
       [1, 4],
       [2, 5]])
```

Reversing an Array

- ➔ np.flip(arr) - **reverse** contents of an array
- ➔ 2-dimensional flipping:
  - ○ Add axis argument to flip rows/columns
    - ■ Axis argument not added: reverse entire array
    - ■ Axis = 0: reverse **rows**
    - ■ Axis = 1: reverse **columns**
  - ○ Reverse along a **specific** column/row
    - ■ Arr[row index]
    - ■ Arr[:, column index]

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

reversed_arr = np.flip(arr)

print('Reversed Array: ', reversed_arr)

Reversed Array:  [8 7 6 5 4 3 2 1]
```

Flattening and Raveling

- ndarray.flatten() - turns multidimensional arrays into 1D arrays
    - Changes made to new 1D array **will not** affect original
- ndarray.ravel() - same as .flatten(), but changes **WILL** affect original array

```
x = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9,
10, 11, 12]])
a1 = x.flatten()
a1[0] = 99
print(x)  # Original array
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
print(a1)  # New array
[99  2  3  4  5  6  7  8  9 10 11 12]
```

FIN