

# Mathematical Operators

- Python operators obey normal **PEMDAS** precedence
  - Expressions are evaluated left to right in your source code
  - Use a single equal sign `=` to assign a value to a variable
  - Use **double** equal signs `==` to test for *equality*
  - Use `*` for multiplication and `/` for division operators
  - Use parenthesis to explicitly override the order of operations
  - Use double asterisks `**` for **exponentiation**

```
celsius = (fahrenheit - 32) * 5 / 9
```

# Open Python Fundamentals

## Introduction to Python

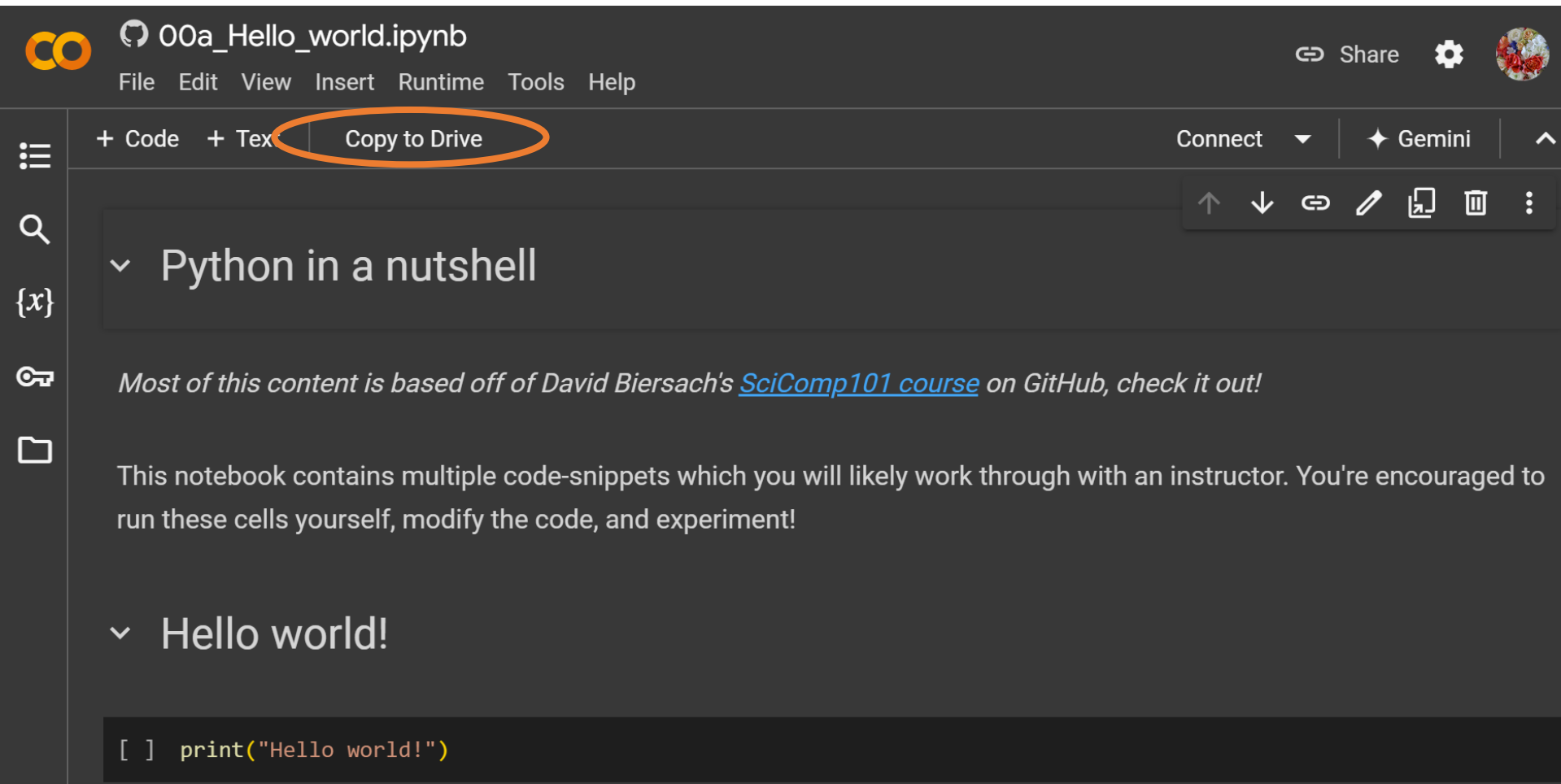
👋 Welcome!

This is the start of the Bootcamp proper. The point of this module is to give you an introduction to the basics of the Python programming language. It assumes you know nothing, but it also assumes that you will "immerse" yourself to some degree. I.e., you need to "google around", use Stack Overflow and just generally have the attitude that you're going to "debug anything and everything" in order to get the most out of this.

## Lecture-guided content

1. [Hello world!](#) - A classic. Contains one line of code. Feel free to use this as scrap too!
2. [Python fundamentals](#) - A brief introduction to some of the Python, and NumPy, fundamentals.
3. [Plotting and more NumPy](#) - Exactly what it sounds like: plotting and more NumPy!
4. [Functions and logic](#) - An introduction to functions and basic Python logic.
5. [Probability and statistics](#) - A basic introduction on how to perform statistics using simple codes in Python, NumPy, etc.
6. [Random number generators and algorithms](#) - The basics of random number generation and algorithms, demonstrated with simple examples.

# Make Your Own Copy



The screenshot shows the Jupyter Notebook interface for a file named '00a\_Hello\_world.ipynb'. The top menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. On the right, there are 'Share', 'Settings', and a user profile icon. Below the menu bar, a toolbar contains '+ Code', '+ Text', and 'Copy to Drive' (which is circled in orange). To the right of the toolbar are 'Connect', 'Gemini', and an expand/collapse icon. The left sidebar shows icons for a menu, search, code editor, and file explorer. The main content area has a section titled 'Python in a nutshell' with a sub-header 'Most of this content is based off of David Biersach's [SciComp101 course](#) on GitHub, check it out!'. Below this is a paragraph: 'This notebook contains multiple code-snippets which you will likely work through with an instructor. You're encouraged to run these cells yourself, modify the code, and experiment!'. Another section titled 'Hello world!' contains a code cell with the text: 

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

# Edit & Run Cells 1...3

You can also  
press  
**SHIFT + ENTER**  
to “run” a cell

The screenshot displays a Jupyter Notebook interface with three code cells. The left sidebar shows icons for a menu, search, variable explorer, keybindings, and file explorer. The top bar has '+ Code' and '+ Text' buttons. The notebook content is as follows:

- Cell 1: Text "Calculate 7 + 2 / 3" (annotated with ①). Below it, a code cell (annotated with ②) contains `# Cell 1` and `print(7 + 2 / 3)`. The output is `7.666666666666667`.
- Cell 2: Text "Calculate (7 + 2) / 3" (annotated with ③). Below it, a code cell (annotated with ④) contains `[2] # Cell 2` and `print((7 + 2) / 3)`. The output is `3.0`.
- Cell 3: Text "Set x to 2 and then print x" (annotated with ⑤). Below it, a code cell (annotated with ⑥) contains `[3] # Cell 3`, `x = 2`, and `print(x)`. The output is `2`.

**P** `()` Parentheses

**E** `X2` Exponents

**M** `×` Multiplication

**D** `÷` Division

**A** `+` Addition

**S** `-` Subtraction

# Identifiers

- Identifiers are just **names** – everything in code has a name
  - Names must be < 64 chars in length
  - They can include upper- or lower-case letters and numbers
  - Identifiers must start with a letter and cannot contain spaces!
- Three types of identifier “casing”
  1. CamelCaseEachWord (first letter is Capitalized)
  2. camelCaseEachWord (first letter is not capitalized)
  3. all **lower\_case** with underscores (Snake case in Python!) ✓
- Identifiers in Python are **case sensitive!!**
  - *x is not the same as X*
  - Never create ALLCAPS identifiers

# Variable Types

- **Variables** store data in memory to be used later
  - Variables can be called whatever you want
  - Pick variable names that mean something to a human
  - Use **snake\_case** (all lower case, underscores to break words)
- Python supports many built-in **data types** for variables:
  - **int** = Stores integers only
  - **float** = Stores Real numbers with 15 digits of precision
  - **bool** = Stores **True** or **False** (Boolean logic, uppercase T/F)
  - **str** = Stores zero or more letters & numbers (a string)
- Python mostly “infers” the type of a variable (**0 vs. 0.0**)

# Displaying Variables

- The **print()** function is used to display the value of variables
- When running inside Thonny, the output will show up in the **Shell** terminal window below your source code
- String *literals* must be enclosed in **double** quotation marks

```
x = 3.14
print(x) → 3.14
print("x") → x
```

# Edit & Run Cells 4...6

The screenshot shows a Jupyter Notebook interface with three code cells. On the left is a sidebar with icons for a menu, search, variables (showing {x}), a key icon, and a file explorer. The top bar has '+ Code' and '+ Text' buttons. The first cell, 'Cell 4', contains the code 'y = 3' and 'print(x, y)', with the output '2 3'. The second cell, 'Cell 5', contains the code 'print(x == y)' and the output 'False'. The third cell, 'Cell 6', contains the code 'print(x \* y == y \* x)' and the output 'True'. Red arrows and circled numbers 1 through 6 point to specific parts of the interface: 1 points to the cell title, 2 points to the print statement, 3 points to the question 'Does x equal y?', 4 points to the double equals sign in the code, 5 points to the cell title 'Demonstrate the commutative property of multiplication', and 6 points to the multiplication expression in the code.

+ Code + Text

Set y to 3 and then print x and y ①

[4] # Cell 4  
y = 3  
print(x, y) ②  
2 3

Does x equal y? ③

[5] # Cell 5  
print(x == y) ④  
False

Demonstrate the commutative property of multiplication ⑤

[6] # Cell 6  
print(x \* y == y \* x) ⑥  
True

You can display multiple variables in a single **print()** statement  
By default, Python will put a space between the values

In Python, we test for equality using **double** equal signs

Multiplication is commutative



# Edit & Run Cells 7...9

The screenshot shows a Jupyter Notebook interface with three code cells. On the left is a sidebar with icons for search, variables, keys, and files. The notebook content is as follows:

- Cell 7:** Title "Demonstrate that subtraction is *not* commutative" (annotated with ①). Code: `# Cell 7`, `print(x - y == y - x)` (annotated with ②). Output: `False`. A purple callout box states: "Subtraction is NOT commutative".
- Cell 8:** Title "Demonstrate that parenthesis override the regular order of operations" (annotated with ③). Code: `[8] # Cell 8`, `print(2 * (x - 5))` (annotated with ④). Output: `-6`. A purple callout box states: "Parentheses override the regular order of operations".
- Cell 9:** Title "Set z to x divided by y, then print z" (annotated with ⑤). Code: `[9] # Cell 9`, `z = x / y` (annotated with ⑥), `print(z)`. Output: `0.6666666666666666`. A purple callout box states: "Expressions (formulas) don't need any numbers".

# Displaying Variables

- Within a **print()** statement, Python can substitute a variable's value into a **placeholder**
  - To make a **placeholder** (aka a *replacement field*) you enclose the variable name between curly braces **{}**
  - Substituting a variable's actual value into its *replacement field* is called *string interpolation*
- Placeholders can also contain **format specifiers**
  - You can specify the number of digits to the right of the decimal, etc.
  - You can specify left/right/center justification, column width, etc.

# print() and f-strings

```
for fahrenheit in range(-44, 217, 4):  
    celsius = (fahrenheit - 32) * 5 / 9  
    print(f"{fahrenheit:>6.2f} F = {celsius:>6.2f} C")
```

Placing a lowercase **f** before the first quote in a **print()** statement indicates you will use some **placeholders**

A placeholder contains the variable's name sandwiched between curly **braces** {}

A **colon** after the variable's name starts a **format specifier**

# Some Common Format Specifiers

<code>:&lt;</code>	Left aligns the result (within the available space)
<code>:&gt;</code>	Right aligns the result (within the available space)
<code>:^</code>	Center aligns the result (within the available space)
<code>:,</code>	Use a comma as a thousand separator
<code>:f</code>	Fix point number format
<code>:n</code>	Number format
<code>:%</code>	Percentage format

**Using format specifiers is optional but makes your output more professional**

# Edit & Run Cells 10...12

**Demonstrate division takes precedence over addition**

[10] # Cell 10  
z = x / y + 0.2  
print(z)

0.8666666666666667

Division before Addition

**Print z using an f-string without any format specifier**

[11] # Cell 11  
print(f"z = {z}")

z = 0.8666666666666667

This placeholder does not contain any format specifier

**Print z using an f-string with 4 digits to the right of the decimal point**

[12] # Cell 12  
print(f"z = {z:.4f}")

z = 0.8667

This format specifier *rounds* to 4 digits to the **right** of the decimal

# Edit & Run age\_in\_weeks.ipynb

The screenshot shows a Jupyter Notebook interface. At the top, there are tabs for '+ Code' and '+ Text'. Below the tabs, the notebook title is 'Convert your age in years to age in weeks, and display both values' (labeled ①). The code cell contains the following Python code:

```
# Cell 1
age_years = 57 ②
age_weeks = age_years * 52 ③

print(f"I am {age_years} years old", end=", ") ④
print(f"which is {age_weeks:,.} weeks.") ⑤
```

The output of the code cell is: 'I am 57 years old, which is 2,964 weeks.'

Type your  
**own** age &  
notice the  
**underscore**

There are  
**52 weeks**  
in a year

`end=", "`  
suppresses  
the line break

The `:,` *format specifier* puts a  
comma between  
every 3 digits

# Edit & Run age\_in\_weeks.ipynb



The screenshot shows a Jupyter Notebook interface. On the left is a sidebar with icons for a menu, search, variables, keys, and files. The main area has a title bar with '+ Code' and '+ Text' buttons. Below the title bar is a heading: 'Convert your age in years to age in weeks, and display both values'. There is a code cell labeled '# Cell 1' with a play button icon and a status indicator '0s'. The code in the cell is as follows:

```
# Cell 1
age_years = int(input("What is your age in years? "))
age_weeks = age_years * 52

print(f"I am {age_years} years old", end=", ")
print(f"which is {age_weeks:,} weeks.")
```

Below the code cell, the output is displayed: 'I am 57 years old, which is 2,964 weeks.'

`input()` - Ask the user for their own age in years!


NOTE – `input` records a STRING by default, so we must force it to record an INTEGER to be able to perform math operations on it

# Extending Python via the **numpy** Package

<https://numpy.org>

[Install](#) [Documentation](#) [Learn](#) [Community](#) [About Us](#) [News](#) [Contribute](#)

# NumPy



The fundamental package for scientific computing with Python

LATEST RELEASE: NUMPY 1.25. VIEW ALL RELEASES

## NumPy 1.25.0 released

2023-06-17

### POWERFUL N-DIMENSIONAL ARRAYS

Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.

### NUMERICAL COMPUTING TOOLS

NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

### OPEN SOURCE

Distributed under a liberal [BSD license](#), NumPy is developed and maintained [publicly on GitHub](#) by a vibrant, responsive, and diverse [community](#).

### INTEROPERABLE

NumPy supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.

### PERFORMANT

The core of NumPy is well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.

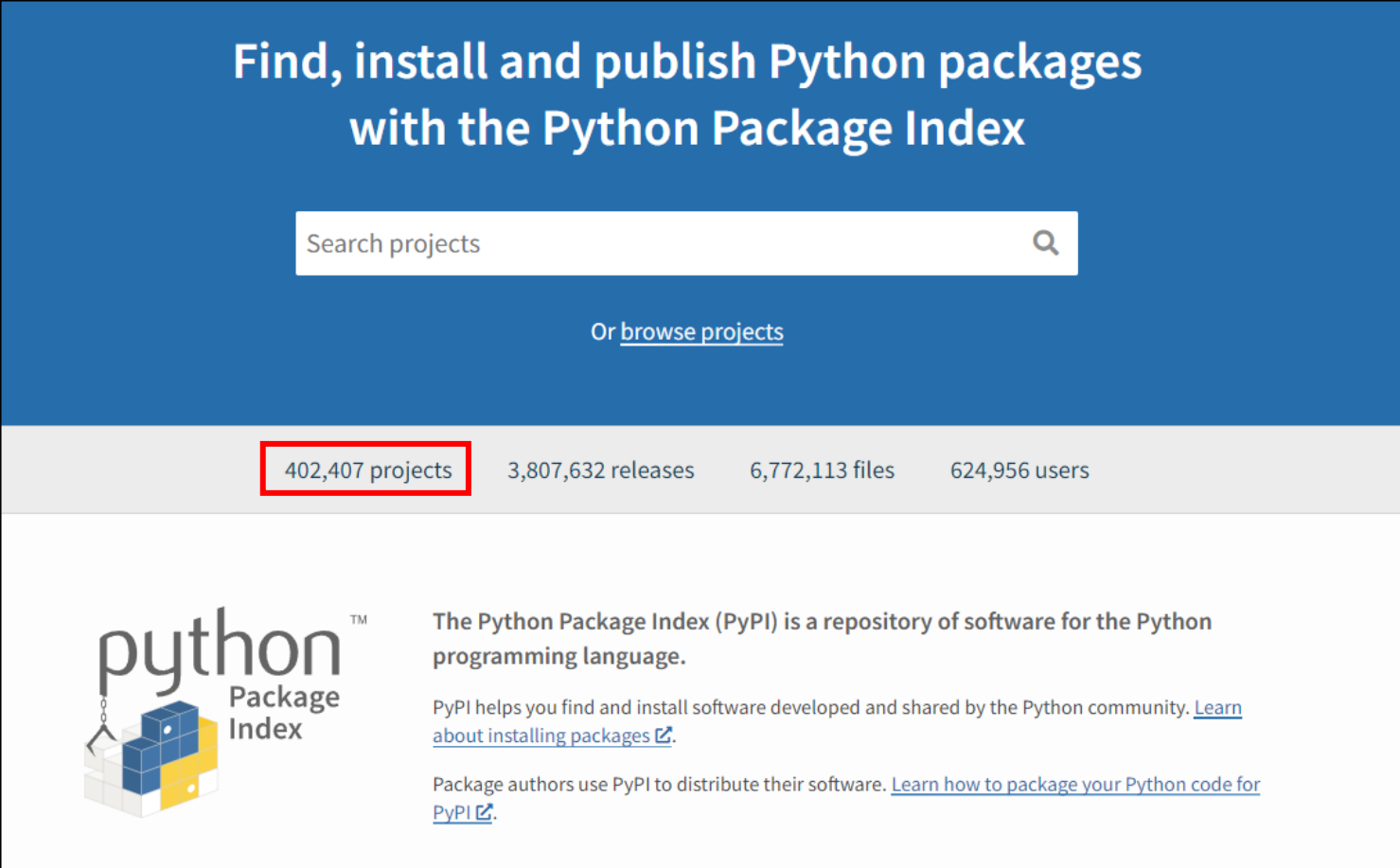
### EASY TO USE

NumPy's high level syntax makes it accessible and productive for programmers from any background or experience level.



# About **PyPI** (Python **P**ackage Index)

<https://pypi.org>




The screenshot shows the PyPI website homepage. At the top, a blue banner contains the text "Find, install and publish Python packages with the Python Package Index". Below this is a search bar with the placeholder text "Search projects" and a magnifying glass icon. Under the search bar, there is a link "Or [browse projects](#)". A light gray bar below the search bar displays statistics: "402,407 projects" (highlighted with a red box), "3,807,632 releases", "6,772,113 files", and "624,956 users". The bottom section features the Python Package Index logo on the left and descriptive text on the right. The text states: "The Python Package Index (PyPI) is a repository of software for the Python programming language." It then explains that PyPI helps find and install software developed and shared by the Python community, with a link to "Learn about installing packages". Finally, it mentions that package authors use PyPI to distribute their software, with a link to "Learn how to package your Python code for PyPI".

Find, install and publish Python packages  
with the Python Package Index

Search projects

Or [browse projects](#)

402,407 projects 3,807,632 releases 6,772,113 files 624,956 users

 **python**™  
Package Index

The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages](#).

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI](#).

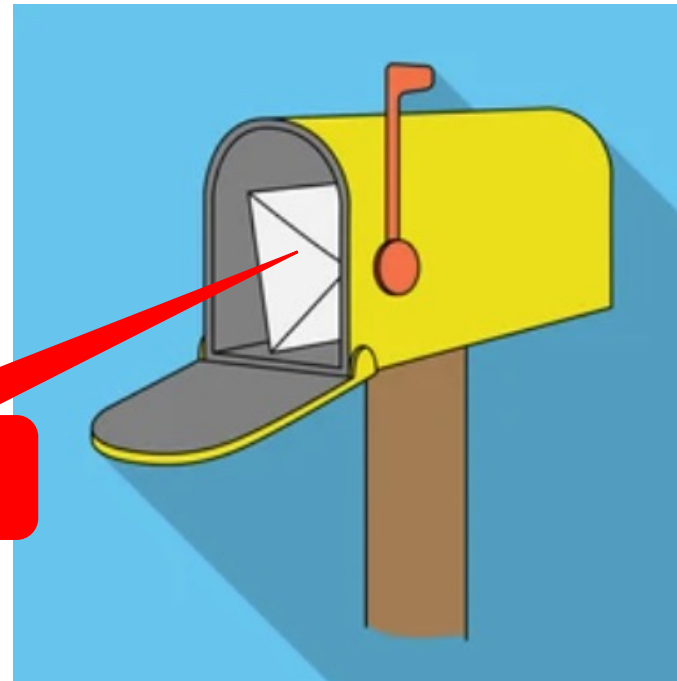
# Numpy Arrays

- An **array** is a set of *elements* having all the same **type**
- An individual element in an array is accessed by using its **index number** within square **[]** brackets
  - Every element has a unique index number
  - No two elements share the exact same index number
  - **The first element has an index = 0**
- The function **size()** returns the *length* of an array, which is the number of elements in the array
- Why is the *last* element in an array at **[size() - 1]**?

# Index Number versus Element Value



`array_name[2]`



# The Bane of All Programmers

- A farmer has a fence 100m long
- He wants to divide it into 100 equal sections
- How many fence posts does he need?

**This problem is why we all agree to always use ZERO as the *first* index value in an array.**

**Remember...**  
**ZERO is a THING!**



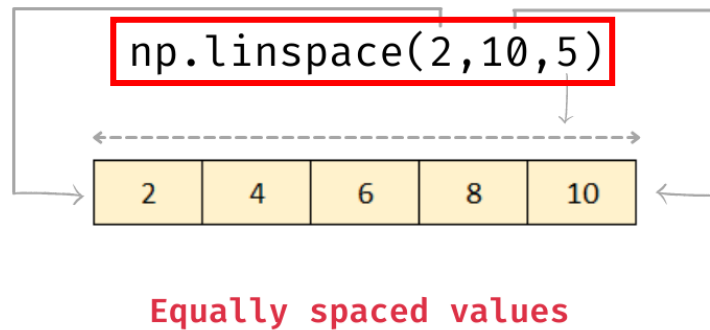
## Off-by-one error

From Wikipedia, the free encyclopedia

An **off-by-one error** (OBOE), also commonly known as an **OBOB** (off-by-one bug), is a [logic error](#) involving the discrete equivalent of a [boundary condition](#). It often occurs in [computer programming](#) when an [iterative loop](#) iterates one time too many or too few. This problem could arise when a programmer makes mistakes such as using "is less than or equal to" where "is less than" should have been used in a comparison or fails to take into account that [a sequence starts at zero rather than one \(as with array indices in many languages\)](#). This can also occur in a [mathematical](#) context.

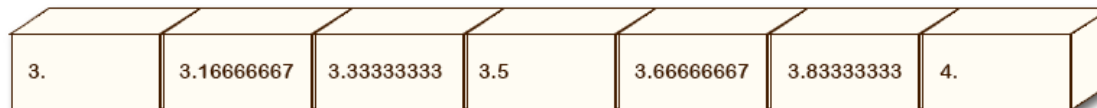
# A Numpy **Linearly Spaced** Array

Creates a "street" of *mailboxes* where the **values** inside are equally spaced between [start, stop]



```
x = np.linspace(3, 4, 7)
```

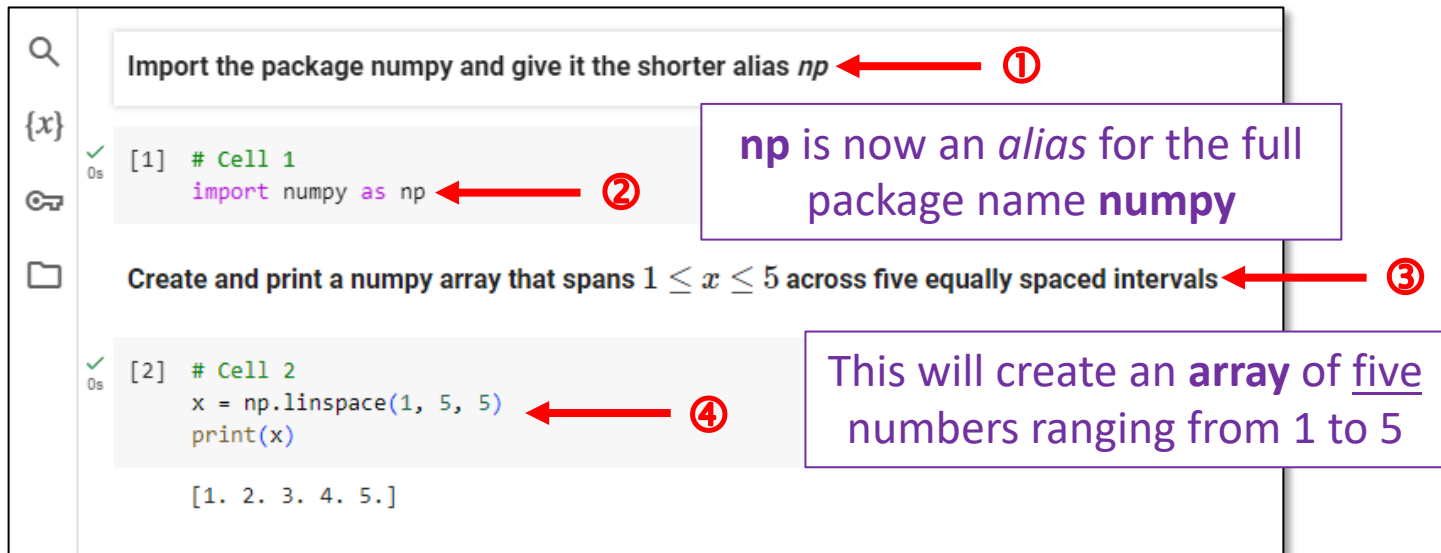
Index  
[0]



Index  
[6]

**np.linspace()** figures out the *step* size based on the range of the linear space and the number of elements you request

# Edit & Run numpy\_arrays.ipynb – Cells 1...2



The screenshot shows a Jupyter Notebook interface with two code cells. The left sidebar contains icons for search, variables, keys, and files. The first cell, labeled '[1] # Cell 1', contains the code `import numpy as np`. A red arrow points from a circled '1' to the text 'Import the package numpy and give it the shorter alias *np*'. A purple box explains that *np* is now an alias for the full package name **numpy**. The second cell, labeled '[2] # Cell 2', contains the code `x = np.linspace(1, 5, 5)` and `print(x)`. A red arrow points from a circled '3' to the text 'Create and print a numpy array that spans  $1 \leq x \leq 5$  across five equally spaced intervals'. Another red arrow points from a circled '4' to the `np.linspace` function call. A purple box explains that this will create an **array of five** numbers ranging from 1 to 5. The output of the second cell is displayed as `[1. 2. 3. 4. 5.]`.

```
Import the package numpy and give it the shorter alias np ①
```

```
[1] # Cell 1
import numpy as np ②
```

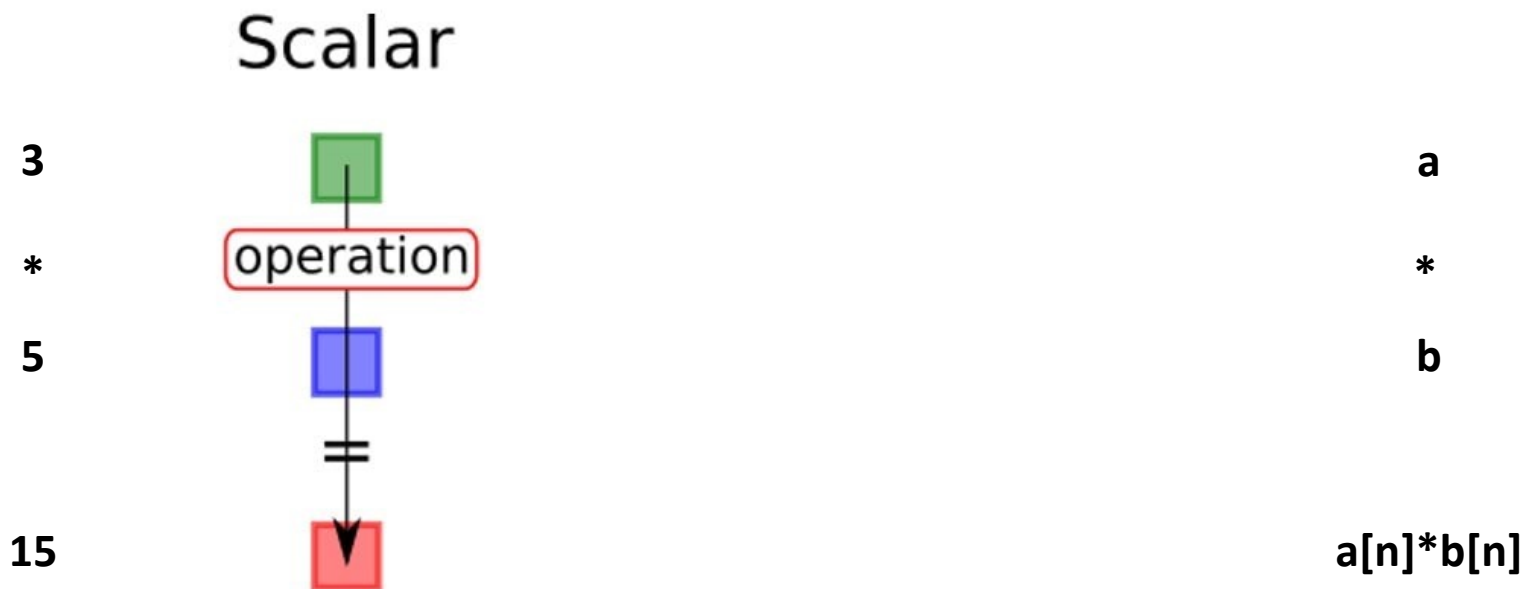
*np* is now an *alias* for the full package name **numpy**

```
Create and print a numpy array that spans  $1 \leq x \leq 5$  across five equally spaced intervals ③
```

```
[2] # Cell 2
x = np.linspace(1, 5, 5) ④
print(x)
```

```
[1. 2. 3. 4. 5.]
```

# Numpy **Vectorized** Operations



A vectorized **scalar** operation applies a function to every element in a *single* array (to each individual cell)

A vectorized **array** operation applies a function to elements in *both* arrays that have the same index value

# Edit & Run numpy\_arrays.ipynb – Cells 3...4

Import the package numpy and give it the shorter alias *np*

[1] # Cell 1  
`import numpy as np`

Create and print a numpy array that spans  $1 \leq x \leq 5$  across five equally spaced intervals

[2] # Cell 2  
`x = np.linspace(1, 5, 5)`  
`print(x)`  
  
[1. 2. 3. 4. 5.]

Display the result of multiplying every element in *x* by 2 ← ①

[3] # Cell 3  
`print(x * 2)` ← ②  
  
[ 2. 4. 6. 8. 10.]

This multiplies each element (individually) in the array “*x*” by 2

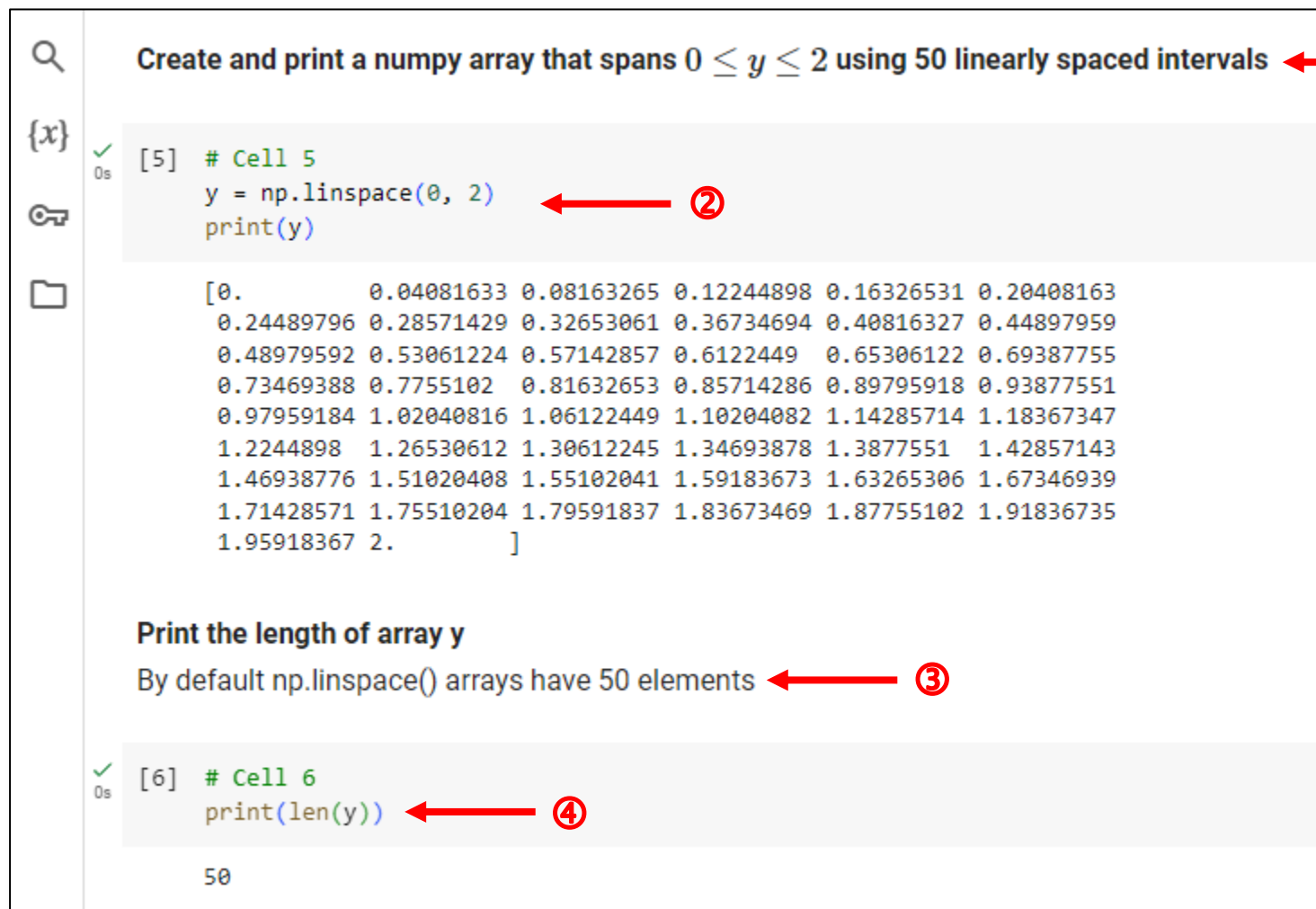
Display the result of squaring every element in *x* ← ③

[4] # Cell 4  
`print(x**2)` ← ④  
  
[ 1. 4. 9. 16. 25.]

This squares each element (individually) in the array “*x*”



## Edit numpy\_arrays.ipynb – Cells 5...6



Search {x} Os Key Folder

Create and print a numpy array that spans  $0 \leq y \leq 2$  using 50 linearly spaced intervals ①

[5] # Cell 5  
y = np.linspace(0, 2)  
print(y) ②

```
[0.          0.04081633 0.08163265 0.12244898 0.16326531 0.20408163
 0.24489796 0.28571429 0.32653061 0.36734694 0.40816327 0.44897959
 0.48979592 0.53061224 0.57142857 0.6122449  0.65306122 0.69387755
 0.73469388 0.7755102  0.81632653 0.85714286 0.89795918 0.93877551
 0.97959184 1.02040816 1.06122449 1.10204082 1.14285714 1.18367347
 1.2244898  1.26530612 1.30612245 1.34693878 1.3877551  1.42857143
 1.46938776 1.51020408 1.55102041 1.59183673 1.63265306 1.67346939
 1.71428571 1.75510204 1.79591837 1.83673469 1.87755102 1.91836735
 1.95918367 2.          ]
```

**Print the length of array y**

By default np.linspace() arrays have 50 elements ③

[6] # Cell 6  
print(len(y)) ④

50

# Edit & Run numpy\_arrays.ipynb – Cells 7...8

**Print the first and last elements in array y** ← ①

We use the square bracket operator to get access to elements via their index number ← ②

The -1 index is a convenience shorthand for the last element in an array ← ③

```
[7] # Cell 7
print(y[0])
print(y[-1])
```

0.0  
2.0

← ④

[-1] is the last element in an array

**Apply the square root operator to every element in array y** ← ⑤

The function `np.sqrt()` is "vector aware"

```
[8] # Cell 8
print(np.sqrt(y))
```

[0. 0.20203051 0.28571429 0.34992711 0.40406102 0.45175395  
0.49487166 0.53452248 0.57142857 0.60609153 0.63887656 0.67005939  
0.69985421 0.72843136 0.75592895 0.7824608 0.80812204 0.83299313  
0.85714286 0.88063057 0.9035079 0.9258201 0.94760708 0.96890428  
0.98974332 1.01015254 1.03015751 1.04978132 1.06904497 1.08796759  
1.10656667 1.12485827 1.14285714 1.16057691 1.17803018 1.19522861  
1.21218305 1.22890361 1.2453997 1.26168012 1.27775313 1.29362645  
1.30930734 1.32480264 1.34011879 1.35526185 1.37023758 1.38505139  
1.39970842 1.41421356]

← ⑥

A **vectorized** operation on a single array

← ⑦  $\sqrt{2} \approx 1.41421356$

# A Shortcut

**Carl Friedrich Gauss**  
(1777 – 1855)

**Sum the integers  
from 1 to 100**



1
2
3
4
5
6
7
8
9
10

1	9
2	8
3	7
4	6
5	
10	

$n = 10$

4 matched rows that each sum to 10

1 row that is  $= 10 / 2 = 5$

1 row that is  $= n = 10$

$$n \left( \frac{n}{2} - 1 \right) + \frac{n}{2} + n = \frac{n * (n + 1)}{2}$$

$= 55$

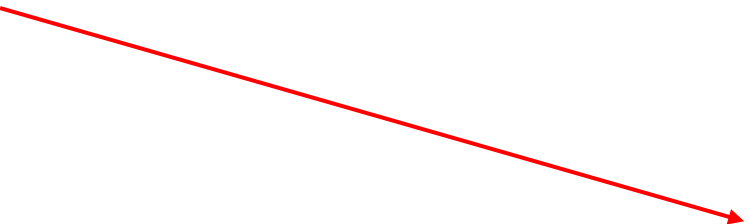
# Another Shortcut

Sum of first  $n$   
*natural* numbers:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2},$$

Sum of squares of first  $n$   
*natural* numbers:

n	n^2	Sum
1	1	1
2	4	5
3	9	14
4	16	30
5	25	55
6	36	91
7	49	140
8	64	204
9	81	285
10	100	385


$$P_n = \sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6} = \boxed{\frac{2n^3 + 3n^2 + n}{6}}.$$

These are functional equations -  
we can now calculate the sums  
*immediately* without having to  
loop over every element!

# Edit gauss\_summation.ipynb – Cells 1..2

① Create a linear space spanning  $1 \leq x \leq 10$  having 10 equally spaced elements

[1] # Cell 1  
import numpy as np  
x = np.linspace(1, 10, 10) ← ②  
print(f"{x = }")

x = array([ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])

Set  $y_1$  to the *cummulative sum* of every element in x ← ③

$$y_1 = \sum_{k=1}^n x_k$$

[2] # Cell 2  
y1 = np.cumsum(x) ← ④  
print(f"{y1 = }")

The numpy cumulative sum function is **vectorized**

y1 = array([ 1., 3., 6., 10., 15., 21., 28., 36., 45., 55.]) ← ⑤

$$\sum_{x=1}^{10} x = 55$$

## Edit gauss\_summation.ipynb – Cells 3..4

Set  $y_2$  to the result of applying the *Gaussian Summation Operator* to each value in  $x$  ← ①

$$y_2 = \frac{x_k(x_k+1)}{2}$$

0s [3] # Cell 3  
y2 = x \* (x + 1) / 2 ← ②  
print(f"{y2 = }")

y2 = array([ 1., 3., 6., 10., 15., 21., 28., 36., 45., 55.])

Does every element in  $y_1 = y_2$ ? ← ③

0s [4] # Cell 4  
print(np.array\_equal(y1, y2)) ← ④

True

When using numpy arrays, most mathematical operators are **vectorized**

$$\sum_{x=1}^{10} x = \frac{x(x+1)}{2} = \frac{10(10+1)}{2} = \frac{110}{2} = 55$$

## Session 02 – Now You Know...

- Python respects **PEMDAS** operator precedence
- How to define variables in Python (**snake\_case**)
- How to use **print()** to show variable values on screen
- The Python **package** called **numpy** (common alias is **np**) provides mathematical functions and support for arrays
- That **np.linspace()** creates an **array** having a given *number of elements* with equal spacing within the **closed-closed** interval **[start, stop]**
- A **vectorized operation** will act on each individual element in one or more arrays and will return the result in a new **array** having the same number of elements as the source

## TASK 02

- Update the code in **age\_in\_seconds.ipynb** to calculate and display your age in both years *and* **seconds**