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

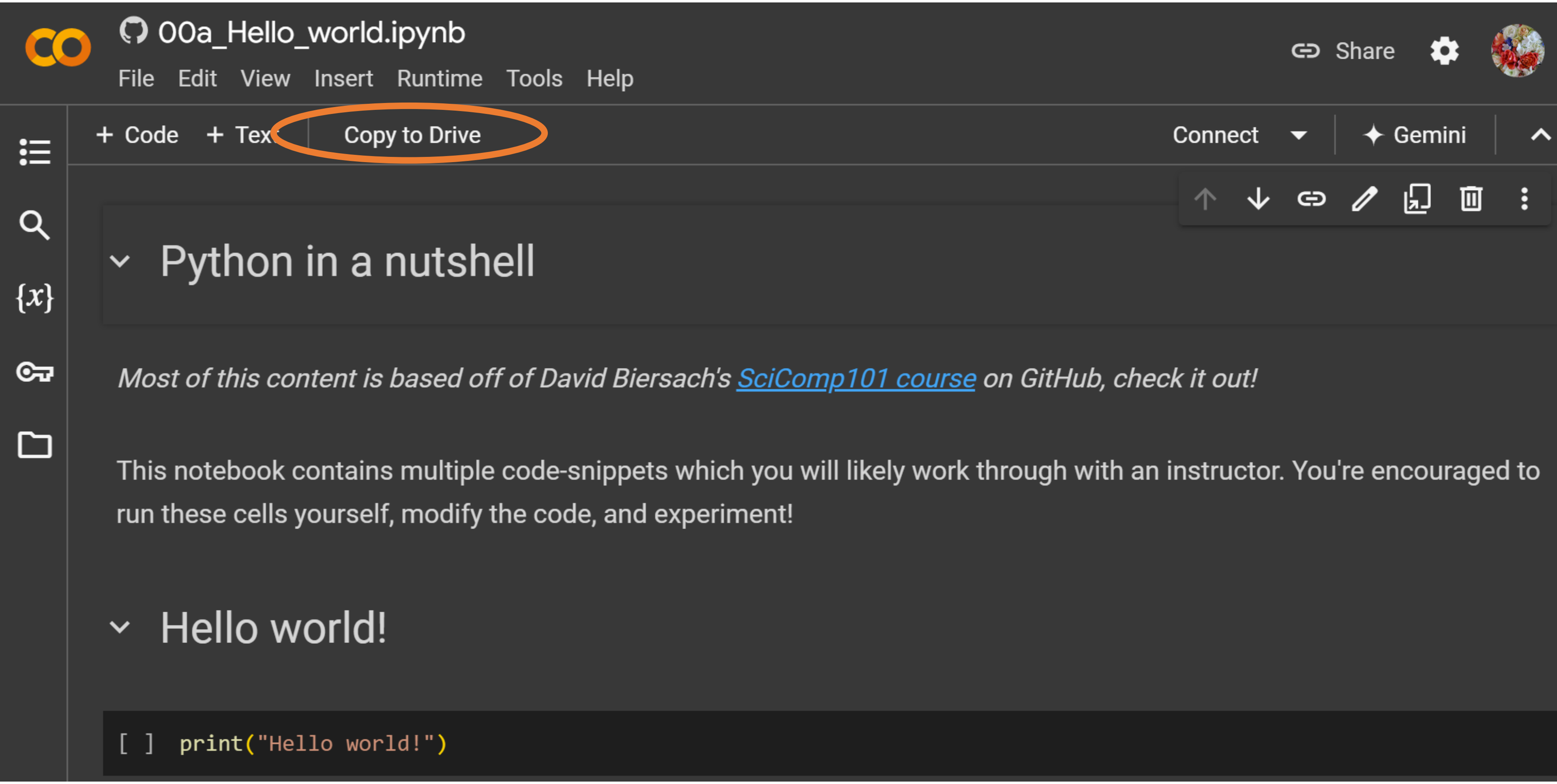
👋 Welcome!

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

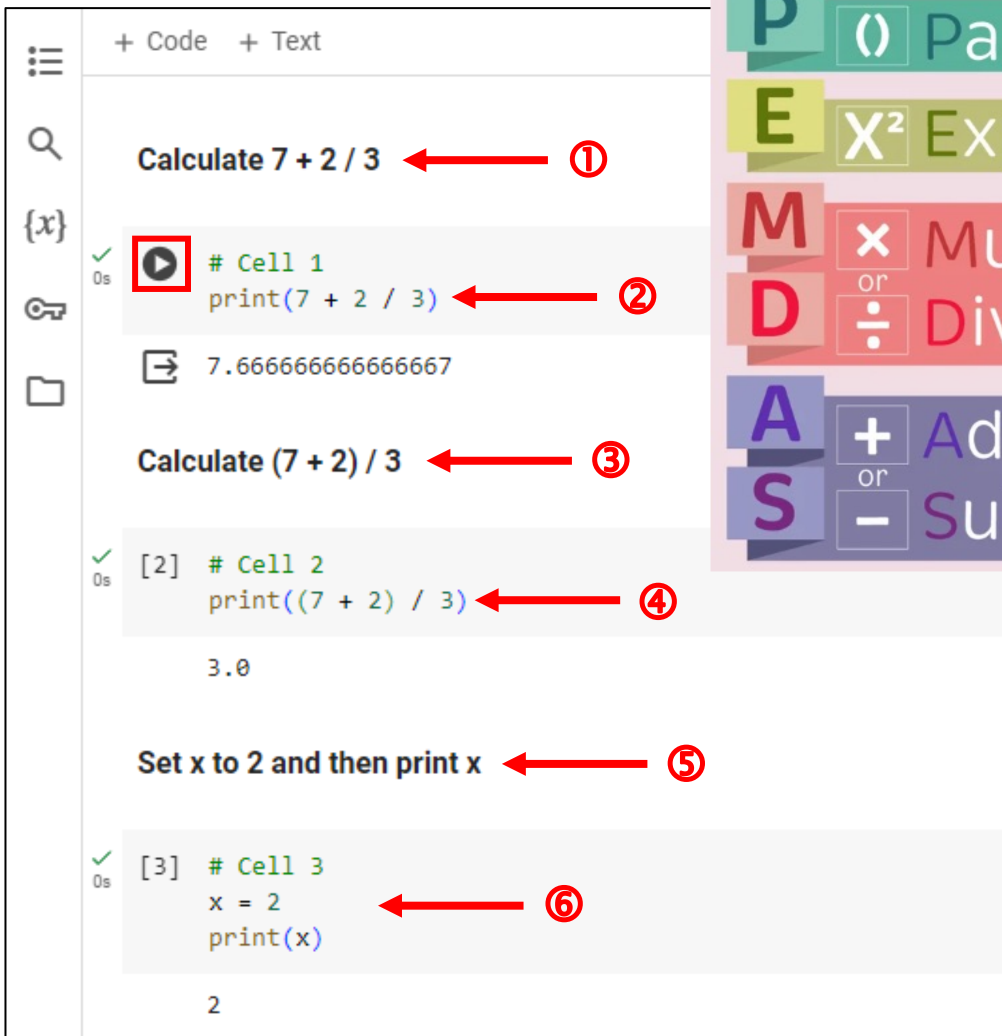
1. Hello world! - A classic. Contains one line of code. Feel free to use this as scrap too!
2. Python fundamentals - A brief introduction to some of the Python, and NumPy, fundamentals.
3. Plotting and more NumPy - Exactly what it sounds like: plotting and more NumPy!
4. Functions and logic - An introduction 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.

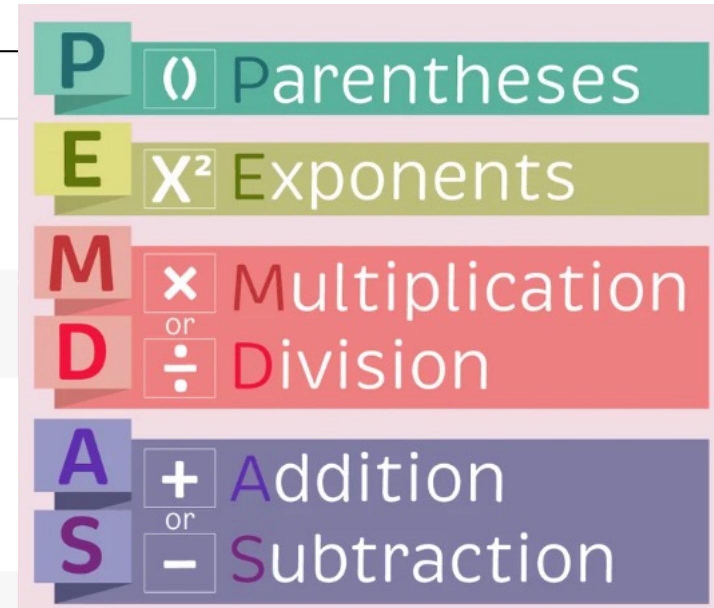
Make Your Own Copy



Edit & Run Cells 1…3

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





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 <u>**cannot contain spaces!**</u>
- Three types of identifier "casing"
  1. CamelCaseEachWord (first letter <u>is</u> Capitalized)
  2. camelCaseEachWord (first letter <u>is not</u> 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 <u>below</u> your source code
- String *literals* must be enclosed in **double** quotation marks

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

```
3.14
x
```

Edit & Run Cells 4…6

+ Code + Text

Set y to 3 and then print x and y ①

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

```
2 3
```

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

Does x equal y? ③

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

```
False
```

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

Demonstrate the commutative property of multiplication ⑤

```
# Cell 6
print(x * y == y * x)
```
⑥

```
True
```

Multiplication is commutative

Edit & Run Cells 7…9

Demonstrate that subtraction is *not* commutative ①

```
# Cell 7
print(x - y == y - x)
```
②

```
False
```

Subtraction is NOT commutative

Demonstrate that parenthesis override the regular order of operations ③

```
[8] # Cell 8
print(2 * (x - 5))
```
④

```
-6
```

Parentheses override the regular order of operations

Set z to x divided by y, then print z ⑤

```
[9] # Cell 9
z = x / y
print(z)
```
⑥

```
0.6666666666666666
```

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

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

Using format specifiers is optional but makes your output more professional

Edit & Run Cells 10…12

+ Code + Text

Demonstrate division takes precedence over addition

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

①

Division before Addition

```
0.8666666666666667
```

Print z using an f-string without any format specifer

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

②

This placeholder does not contain any format specifier

```
z = 0.8666666666666667
```

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

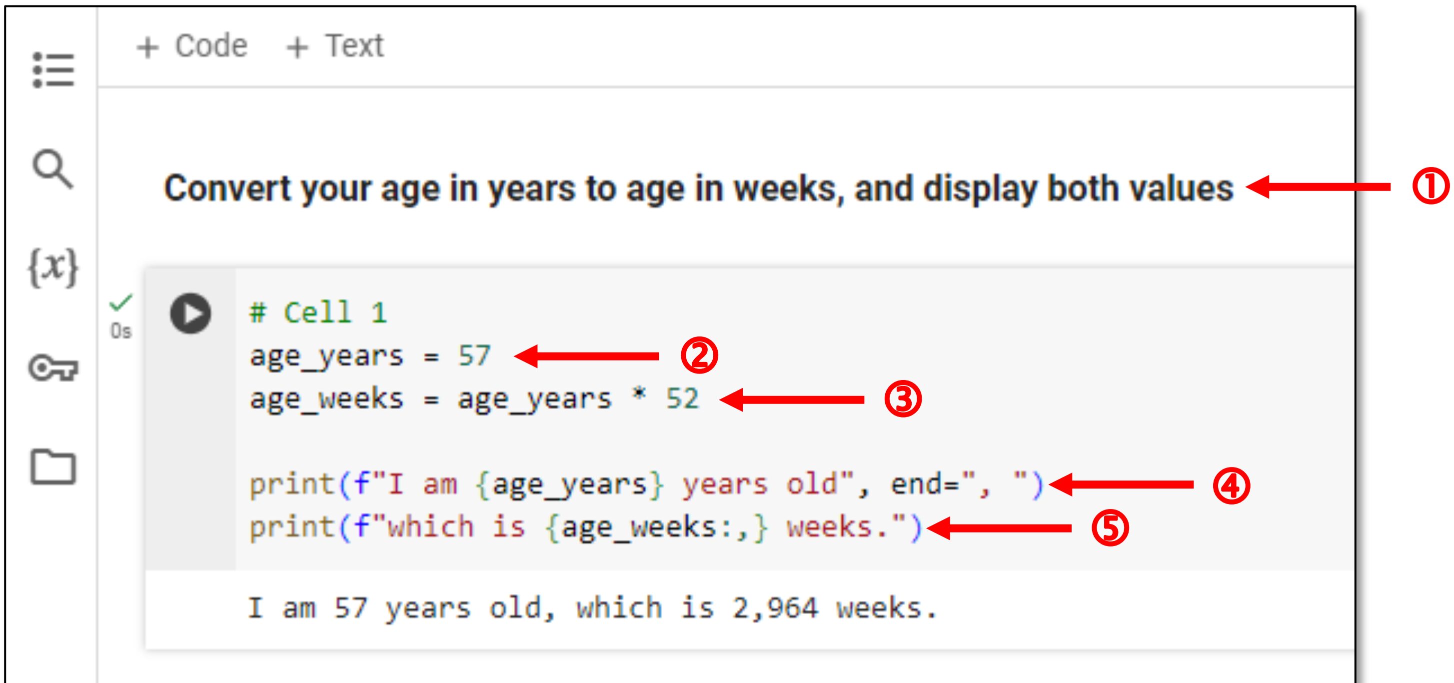
```
[12] # Cell 12
     print(f"z = {z:.4f}")
```

③

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

```
z = 0.8667
```

Edit & Run age_in_weeks.ipynb



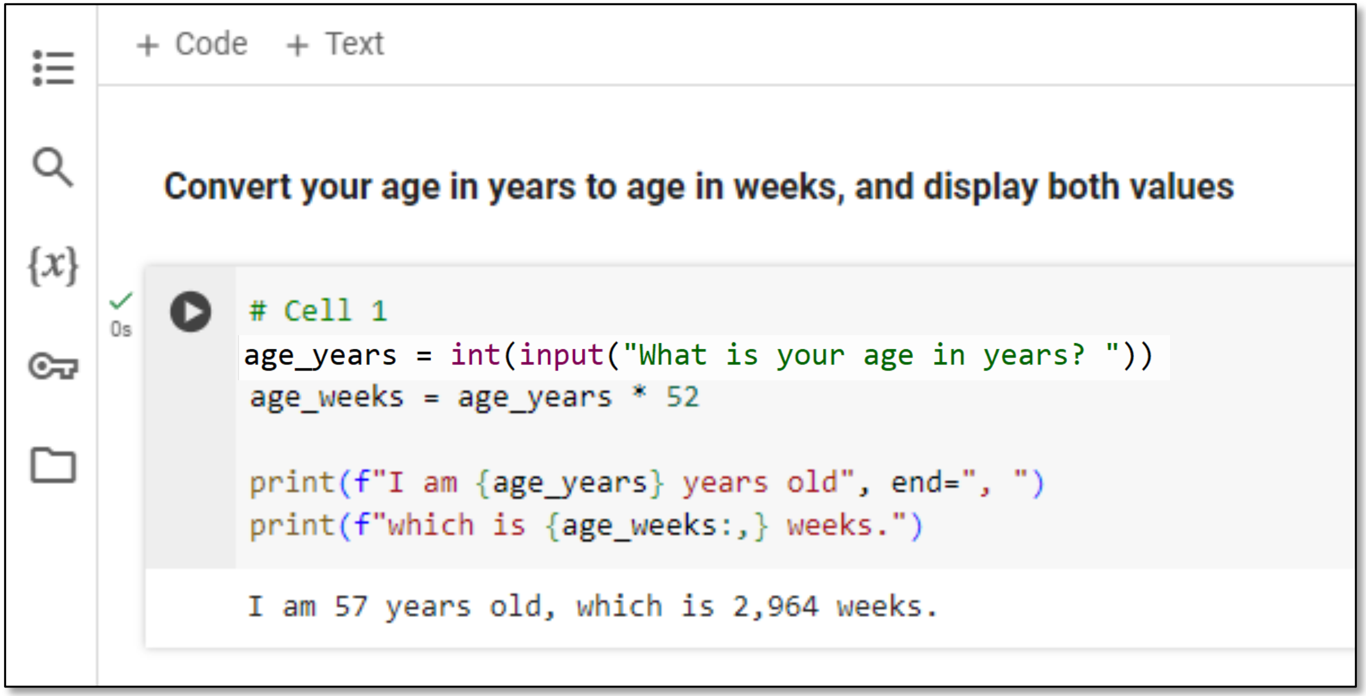
Type your **own** age & notice the **underscore**

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

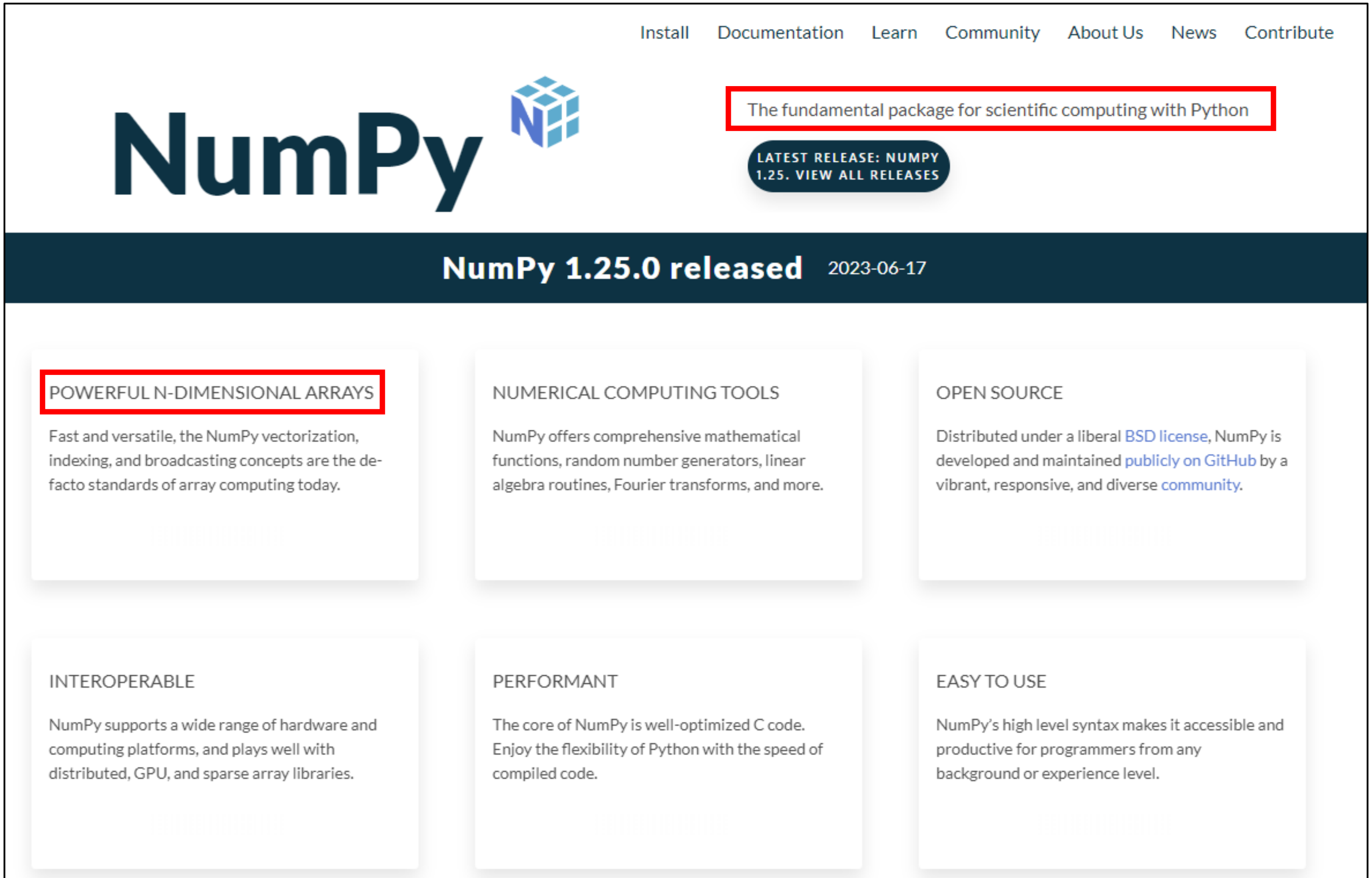


input() - Ask the user for their own age in years!

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

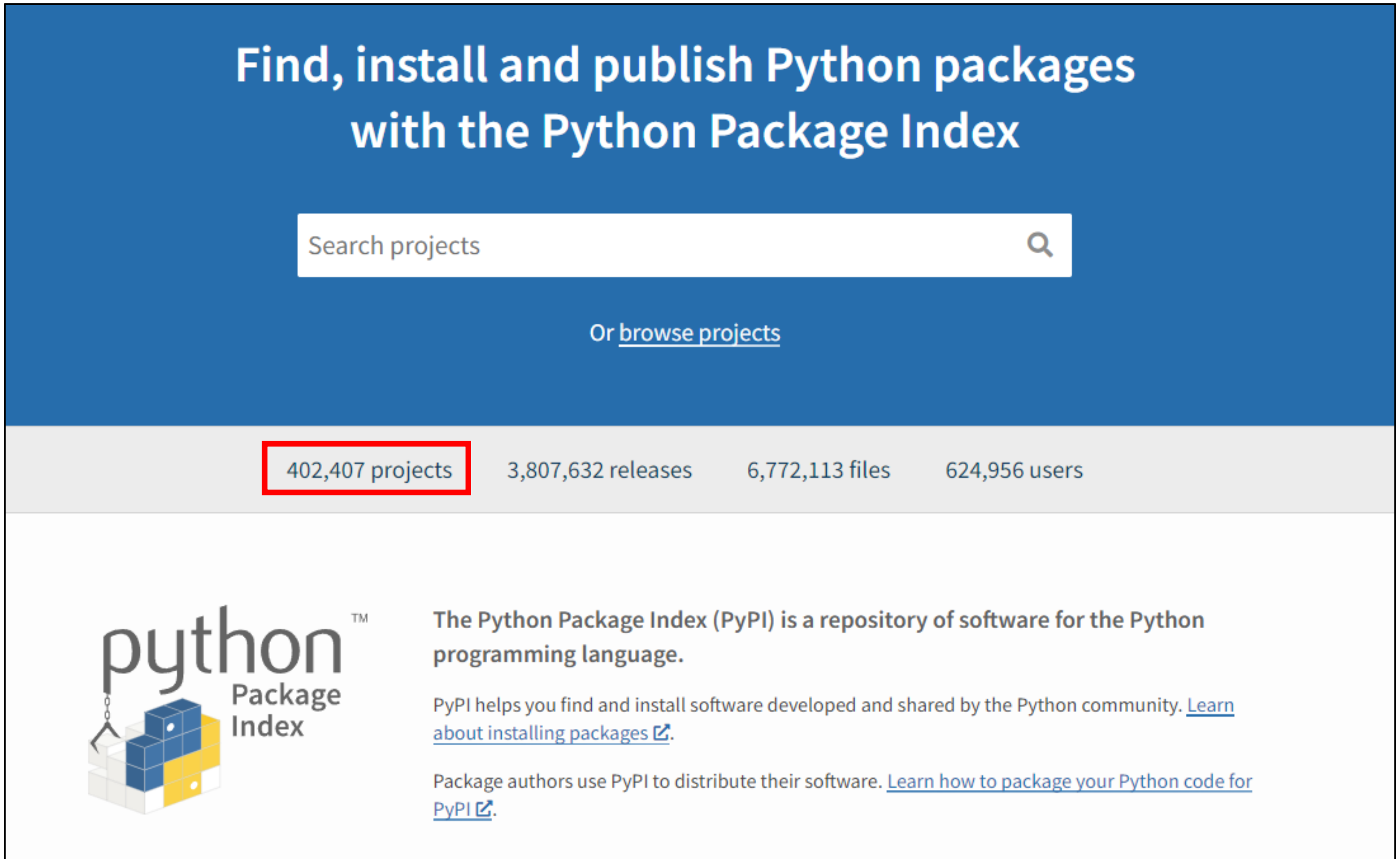
Extending Python via the **numpy** Package

https://numpy.org



About **PyPI** (Python **Package** Index)

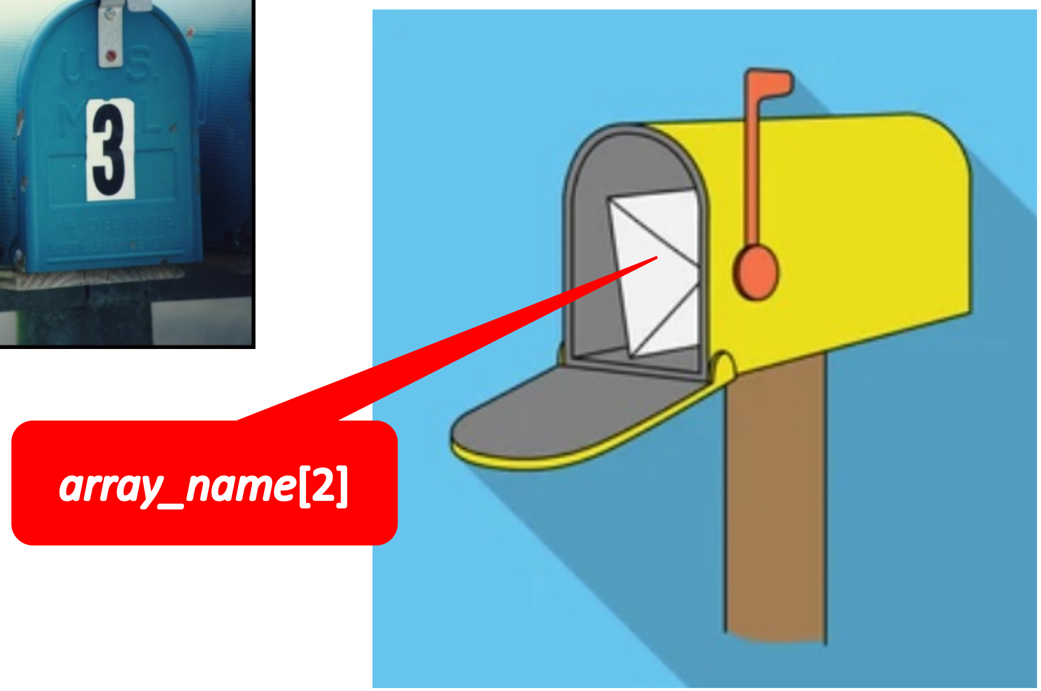
https://pypi.org



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**



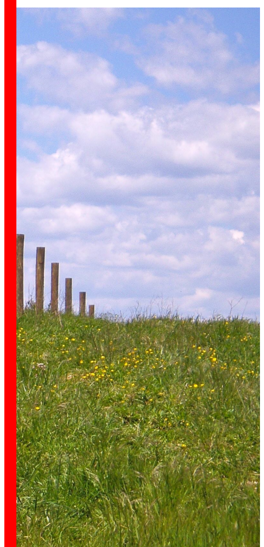
The Bane of All Programmers

- A farmer
  100m lon
- He wants
  each fenc
- How mar
  need?

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

Remember…

ZERO is a THING!



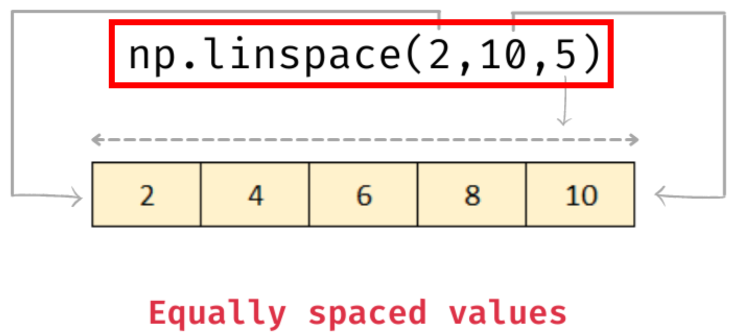
Off-by-one err

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.

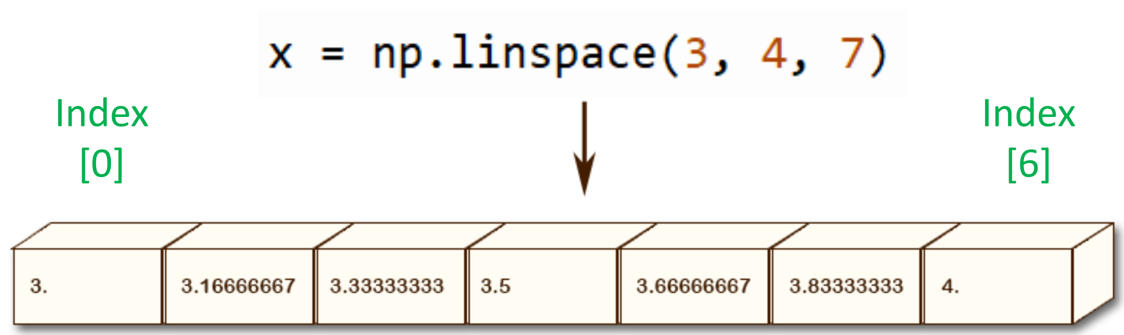
A Numpy **Linearly Spaced** Array

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



Index [0]

Index [6]



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

Edit & Run numpy_arrays.ipynb – **Cells 1…2**

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**

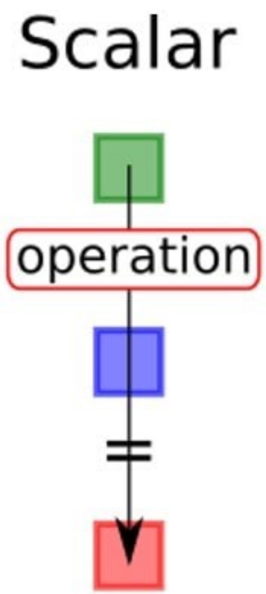
Create and print a numpy array that spans $1 \leq x \leq 5$ across five equally spaced intervals ③

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

     [1. 2. 3. 4. 5.]
```
④

This will create an **array** of five numbers ranging from 1 to 5

Numpy **Vectorized** Operations



3 | a

* | *

5 | b

15 | a[n]*b[n]

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 <u>index</u> 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.  4.  6.  8. 10.]
```

2

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)

     [ 1.  4.  9. 16. 25.]
```

4

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

Edit numpy_arrays.ipynb – **Cells 5…6**

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])
```
④

[-1] is the **last** element in an array

```
0.0
2.0
```

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))
```
⑥

A **vectorized** operation on a single array

```
[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]
```
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

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

= 55

Another Shortcut

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

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

Sum of <u>squares</u> 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 <u>functional equations</u> - we can now calculate the sums *immediately* without having to loop over every element!

Edit gauss_summation.ipynb – **Cells 1…2**

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

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

```
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)
    print(f"{y1 = }")
```
④

The numpy cumulative sum function is **vectorized**

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

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

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

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

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

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

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

Does every element in $y_1 = y_2$? ③

```
[4] # Cell 4
    print(np.array_equal(y1, y2))        ④
```

```
True
```

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

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 <u>arrays</u>
- 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**