Statements and Scopes

- A **statement** does something (think: *sentence*)
  - A statement is either a declaration, a keyword, or a function
  - Statements are executed from top to bottom of a scope
- A **scope** contains one or more statements (think: *paragraph*)
  - A scope in Python begins with a **colon :**
  - Scopes are also denoted via **indentation**
  - All the statements in a scope **must** start at the same **column**
  - Scopes can be *nested* – each inner scope is further indented
  - Certain Python statements "introduce" (**require**) a new scope
  - In Python, white space is significant – indentation matters!

Statements and Scopes

A **for** loop introduces a scope with a **colon :**

```
for n in range(2, 10_000):
    if is_perfect(n):
        print(n)
```

An **if** statement introduces a scope with a **colon :**

Statements within a scope are **indented**

Defining a function

def is how we define a function

The function name

The **inbound** *parameters*

A function introduces a scope

```
def is_perfect(n):
    x = np.arange(1, n)
    factors = x[np.where(n % x == 0)]
    return np.sum(factors) == n
```

Functions can return a value

Functions are just "named" scopes!

The most <u>important</u> function: **main**()

Functions are not executed (run) until they are explicitly called.
The **def** keyword merely **defines** a function but does **NOT** run it

By convention, programmers use **main()** as the name of the **<u>first</u>** function they want the computer to call when starting the program, and it typically appears near the end of the source code file

This is the **first line of executable code** in the program, and its sole purpose is to call your custom **main()** function

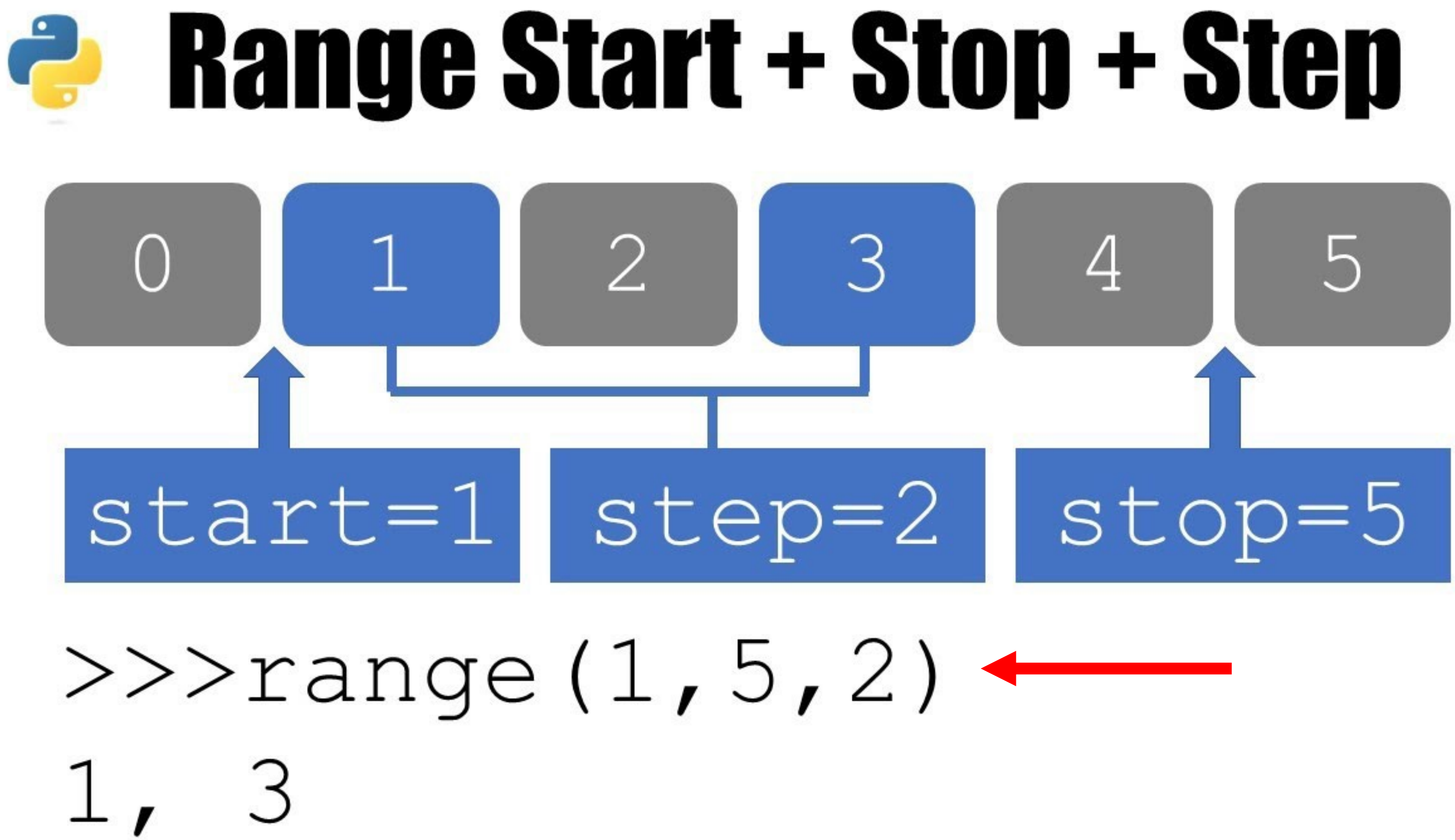
```python
# perfect_numbers.py

import numpy as np


def is_perfect(n):
    x = np.arange(1, n)
    factors = x[np.where(n % x == 0)]
    return np.sum(factors) == n


def main():
    for n in range(2, 10_000):
        if is_perfect(n):
            print(n)


main()
```

Your program now starts at the first statement in **main()**

Use range() to Create a Sequence

- It is common to use the **range**() function to generate a sequence of numbers within some interval
- The **range**() function takes three parameters: (**start**, **stop**, **step**)
  - The **stop** value is required but the **start** and **step** values are *optional*
  - The *default* value (if unspecified) for **start** is **0** and for **step** is **1**
  - The range is inclusive, *exclusive*: **[start, stop)**

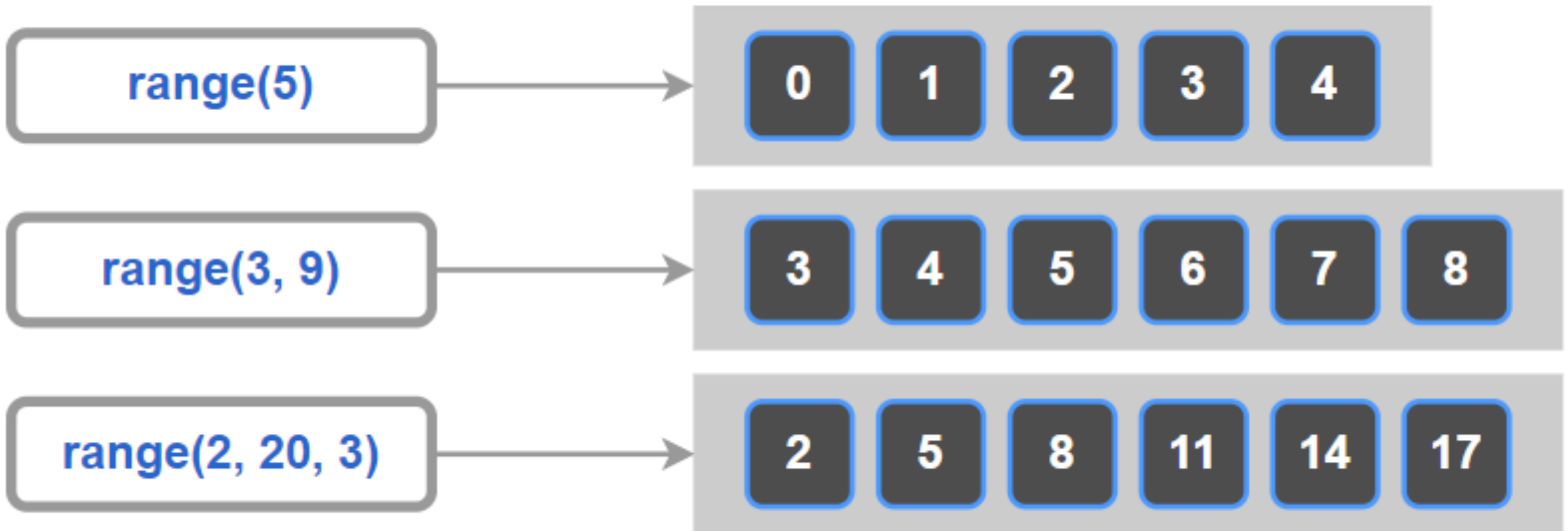
Use **range()** to Create a Sequence



Use range() to Create a Sequence

	Syntax	Output
Single Argument	`range(j)`	`0,1,2,3,4,…., j-1`
	Ex: `range(10)`	`0,1,2,3,4,5,6,7,8,9`
Double Argument	`range(i, j)`	`i,i+1,i+2,i+3,…,j-1`
	Ex: `range(1, 10)`	`1,2,3,4,5,6,7,8,9`
Triple Argument	`range(i, j, k)`	`i,i+k,i+2k,i+3k,…,j-1`
	Ex: `range(1, 10, 2)`	`1,3,5,7,9`

Use **range()** to Create a Sequence



for loops

- A **for** loop executes all the statements within its scope for *each* number within a given range
- It is common to use the **range**() function to create the array of numbers passed into the **for** statement
- The **range**() function takes three parameters: (**start**, **stop**, **step**)
  - The **stop** value is required but the **start** and **step** values are *optional*
  - The *default* value (if unspecified) for **start** is **0** and for **step** is **1**
  - The range is inclusive, *exclusive*: **[start, stop)**

for loops

```python
# Example for range() function

for item in range(0,10):
    print(item, end = ' ')
```

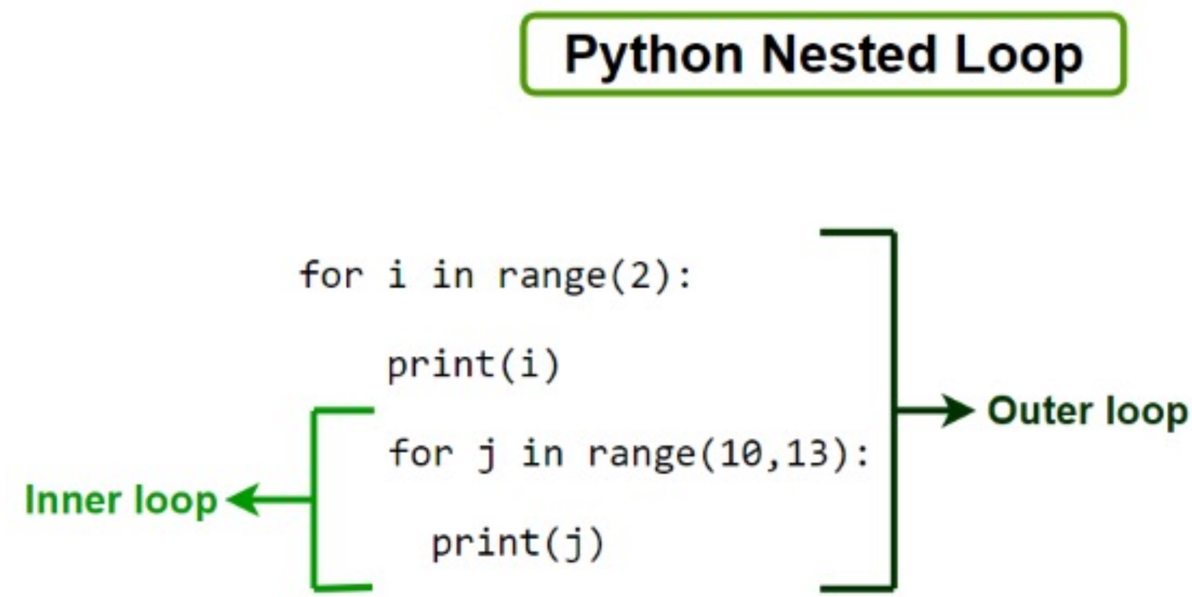
for loops

```python
# Example for range() function

for item in range(0,10):
    print(item, end = ' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

```python
for item in range(10,0,-2):
    print(item, end = ' ')
```

for loops


Python Nested Loop
for i in range(2):
print(i)
for j in range(10,13):
print(j)
Outer loop
Inner loop

if Statement

- An **if** statement identifies which code block (scope) to run based on the value of a **Boolean expression**
  - The expression (the *condition*) **must evaluate** to *either* a **True** or a **False** value
  - If the condition is **True**, then the scope immediately following the **if** statement is executed
  - If the condition is **False**, and there is an **else** clause, then the scope immediately following the **else** statement is executed
  - Every **if** statement does not need to have an **else** clause
  - You can also use **elif** to continue to test for other conditions
- The **if**, **elif**, and **else** statements *all* introduce scopes (meaning they require a **colon** at the end of their line)

if Statement

def is how we define a function

```
if test_expression1:
    #body of if
    statement(s)
elif test_expression2:
    #body of elif_1
    statement(s)
elif test_expression3:
    #body of elif_2
    statement(s)
else:
    statement(s)
```

```
def get_capital(country):
    if country == 'India':
        return 'New Delhi'
    elif country == 'France':
        return 'Paris'
    elif country == 'UK':
        return 'London'
    else:
        return None
```

Be mindful of the **colons :** that start the scopes

Functions can return values

if Statement

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

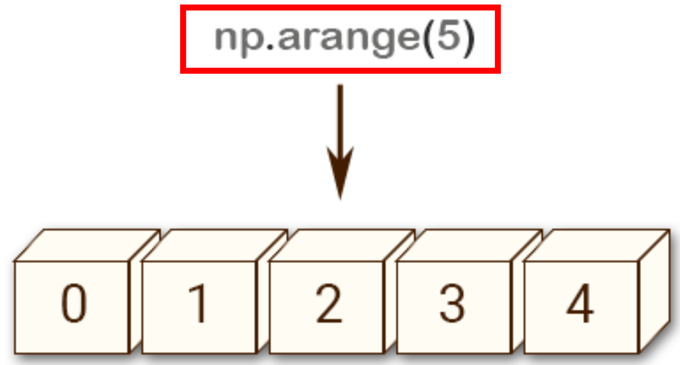
Break Statement

Body of the For Loop

The **break** statement immediately exits from the closest enclosing loop structure

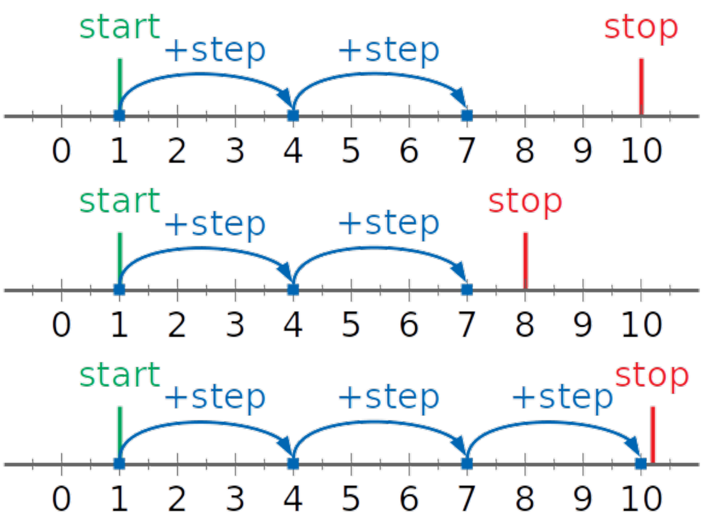
Create a **Numpy** Array from a Range

Creates a "street" of *mailboxes* where the **value** inside each mailbox follows the requested **range**



Just like **range()** the *default* **start** value is **0** and the *default* **step** value is **1**, and the stop value is <u>exclusive</u>

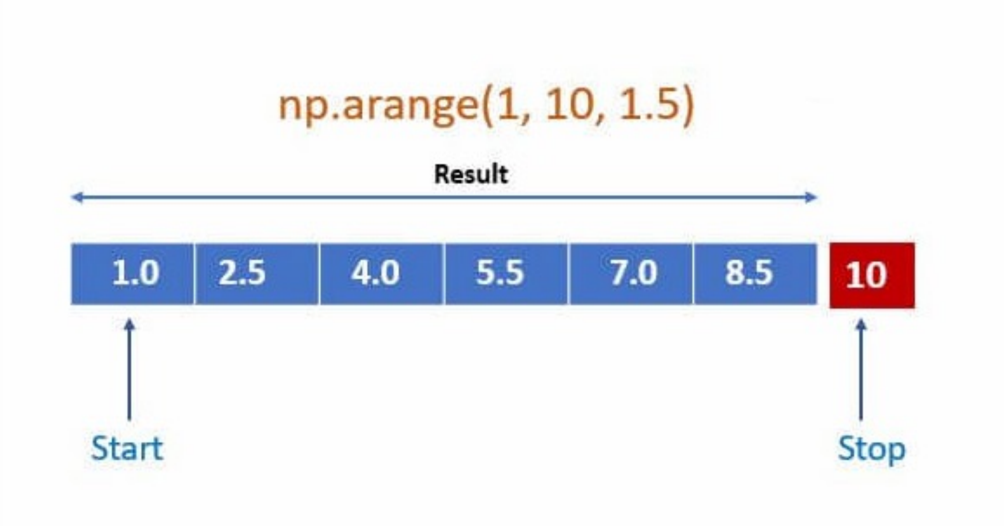
```
>>> np.arange(1, 10, 3)
array([1, 4, 7])

>>> np.arange(1, 8, 3)
array([1, 4, 7])

>>> np.arange(1, 10.1, 3)
array([1., 4., 7., 10.])
```



Create a **Numpy** **Array** from a **Range**

```
np.arange(1, 10, 1.5)
```

Create a **Numpy Array** from a **Range**




```
a = np.arange(1, 11)
```

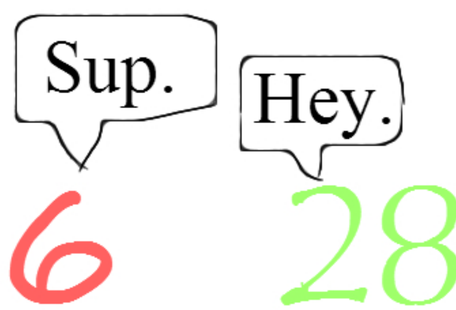
The Modulus (%) Operator

- The "**mod**" operator (**%**) returns the integer **remainder** of an implicit division operation, e.g., **37 % 5 = 2**
- Use double equals operator (**==**) when testing for **equality**

```
def is_perfect(n):
    x = np.arange(1, n)
    factors = x[np.where(n % x == 0)]
    return np.sum(factors) == n
```

Perfect Numbers

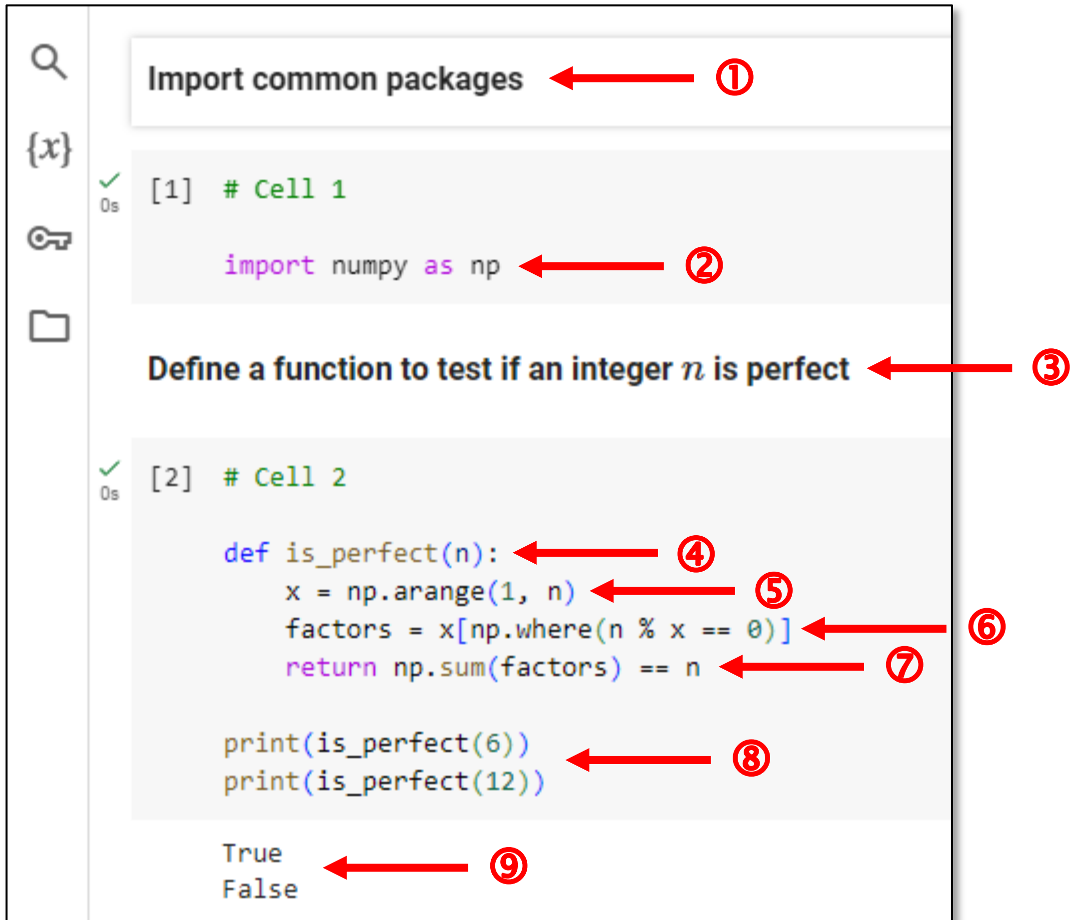
- Write a program to calculate and display all the perfect numbers **n** $(n \in \mathbb{Z}^{+})$ between **2** and **10,000** (inclusive, exclusive)
- An integer **n** is **perfect** when the sum of its *proper* divisors (all divisors including **1,** but <u>not</u> including **n**) is equal to **n**
- Example: **6 = 1 + 2 + 3**



Perfect Numbers

Number	Positive Factors	Sum of all factors excluding itself
1	1	0
2	1, 2	1
3	1, 3	1
4	1, 2, 4	3
5	1, 5	1
$\boxed{6}$	1, 2, 3, 6	$\boxed{6}$ Perfect!
7	1, 7	1
8	1, 2, 4, 8	7
9	1, 3, 9	4
10	1, 2, 5, 10	8
11	1, 11	1
12	1, 2, 3, 4, 6, 12	16

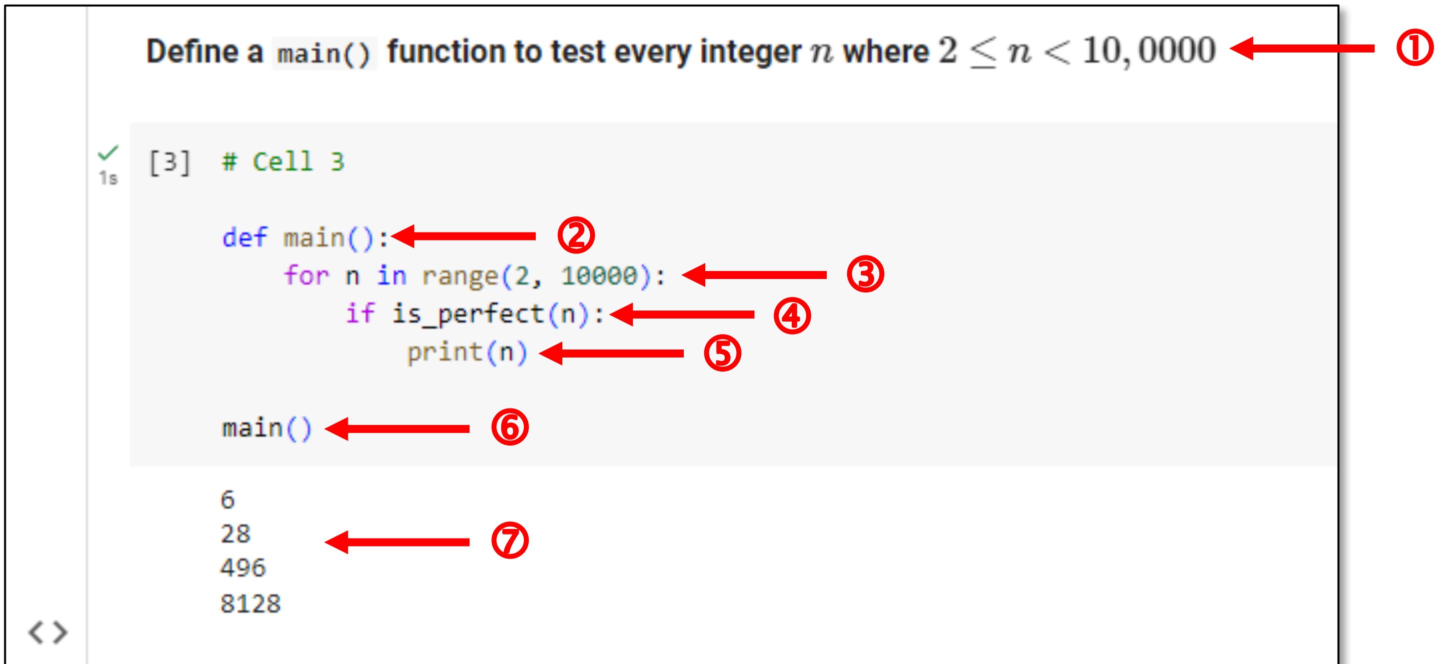
Edit perfect_numbers.ipynb – **Cells 1…2**



np.where() returns all the index numbers within an array where the condition is true for any of its elements

factors is now an array containing only the elements in **x** where the condition was true

Edit perfect_numbers.ipynb – **Cells 3**



It is not typical in Jupyter Notebooks (.ipynb) to define a **main()** function

However, it is very common to define a **main()** in stand-alone Python scripts (.py)

Code inside a function (within a **def:** scope) is not executed until it is explicitly invoked (called)

This is true even for code inside the *special function* named **main()** – you still must explicitly call **main()**

Run perfect_numbers.py

```
6
28
496
8128
```

Do you notice anything special about these numbers?

Bonus points: Given a perfect number **n**, what is the **sum** of the **reciprocals** of its divisors (including **1** *and* **n**) ?

Perfect Numbers

Euclid–Euler theorem

$$\{p, (2^p - 1)\ \} \in primes \rightarrow 2^{(p-1)}(2^p - 1)\ is\ perfect$$

p	2^p-1	n
2	3	6
3	7	28
5	31	496
7	127	8,128
11	2,047	~~2,096,128~~
13	8,191	33,550,336
17	131,071	8,589,869,056

2047 = 23 x 89

Note: This formula might not be the only one to generate perfect numbers – we still don't know for sure!

while Loop

- A **while** loop executes all the statements within its scope as long as the loop conditional remains **True**

The loop conditional

```
t = 10
while t > 0:
    print(t)
    t = t - 1
```

A **while** loop introduces a scope

while Loop

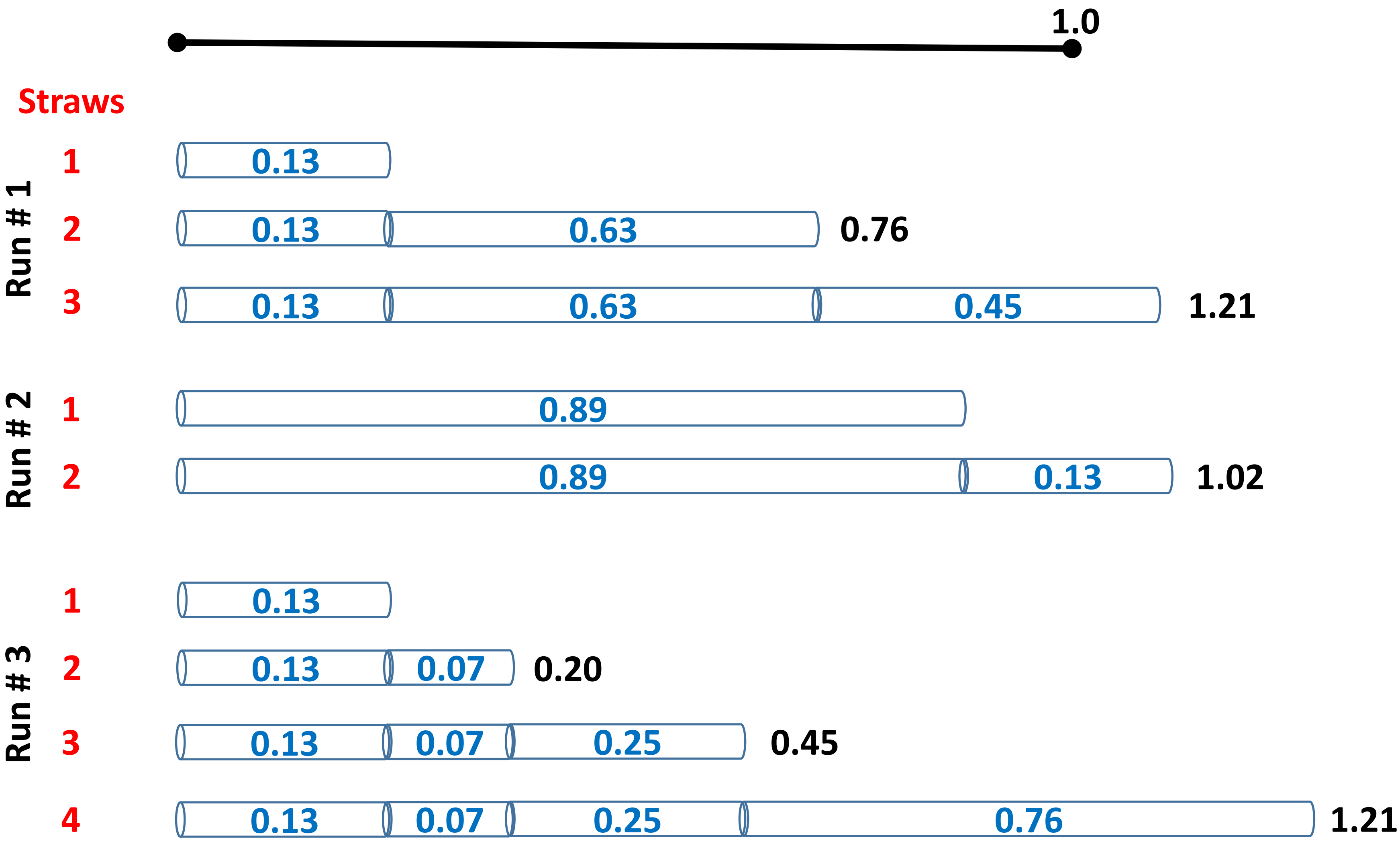
```
i = 5
while i > 0:
    i = i -1
    if i == 2:
        break
    print("inside loop", i)
else:
    print('Inside else')
```

Random Straws

- Write a program to perform **one million runs** of an experiment that places a varying number of straws ***end-to-end*** on each run
- In each run, start with a single straw of **random** length between $0 < n \leq 1$
- Then enter a loop that keeps adding additional straws of **random** length ($0 < n \leq 1$) until the total length is > 1
- Find the **mean** number of straws added before the total length > 1, across *all* million runs of the experiment

https://www.romper.com/parenting/toddler-lines-up-toys

Random Straws



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

Import common packages ①

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

Define a function to perform one run (one trial) of the random straws experiment ③

```
[2] # Cell 2
    def run_trial():  ④
        total_length = 0.0
        num_straws = 0  ⑤
        while total_length <= 1.0:  ⑥
            total_length += (1 - np.random.rand())  ⑦
            num_straws += 1  ⑧
        return num_straws


    print(run_trial())  ⑨
```

```
5
```
⑩

x += 1
is the same as
x = x + 1

np.random.rand()
returns a random
number **[0,1)**

Edit random_straws.ipynb – **Cell 3**

Define and call a `main()` function to run one million trials

```
[3] # Cell 3
    def main():                                   ①
        trials = 1_000_000                        ②
        straws = 0                                ③

        for _ in range(trials):                   ④
            straws += run_trial()                 ⑤

        print(f"{straws / trials:0.5f}")          ⑥
        print(f"{np.e:0.5f}")                     ⑦


    main()                                        ⑧

    2.71854
    2.71828                                       ⑨
```

The **underscore _** symbol is the anonymous ("I don't care") variable placeholder

Computing with Random Numbers?

2.71854
2.71828

We just estimated the base of the natural logarithm using *nothing* but **random numbers**!

$$e = \lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = 2.718281828459\ldots$$

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

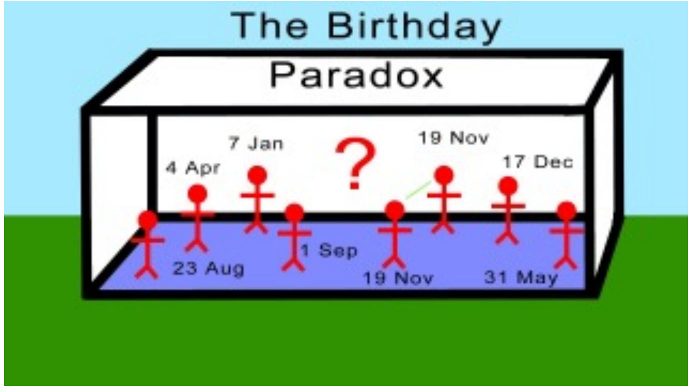
We can *estimate* complicated calculations using nothing but **random numbers**!

Leonhard Euler
(1707 – 1783)

Euler **noticed things** that many others did not…

Birthday Paradox

- Given a class size of **n** students, write a program to calculate the ***probability*** that at least two students in that class share the same birthday
- Your code should calculate this probability for **10,000** classes, each between **2** and **80** students inclusive
- Assume there is only 365 days in a year (no leap years)
- What is the **minimum** required class size to have **> 50%** probability of two similar birthdays?



Run birthday_paradox.ipynb – **Cells 1...3**

Note: You should not edit this file!

Import common packages

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
     from numba import njit                         ①
```

Declare two `GLOBAL` variables ②

Using **GLOBAL** variables in a standalone Python script/program is discouraged but all too common in Notebook files

```
[2]  # Cell 2
     total_classes = 10_000                         ③
     max_size = 80                                  ④
```

Define an `numba` accelerated function to test if any two students within a given class size share the same birthday ⑤

```
[3]  # Cell 3
     @njit                                          ⑥
     def shared_birthdays(class_size):              ⑦
         b = np.random.randint(0, 365, class_size)  ⑧
         for i in range(b.size - 2):
             for j in range(i + 1, b.size):         ⑨
                 if b[i] == b[j]:                   ⑩
                     return True
         return False


     shared_birthdays(class_size=20)                ⑪

     True                                           ⑫
```

Run birthday_paradox.ipynb – **Cell 4…5**

Define an `numba` accelerated function to calculate the probability of having at least one shared birthday in 10,000 random classes of size ranging from 2 to 80 inclusive ①

```
[11] # Cell 4
     def calc_probabilities():            ②
         p = np.zeros(max_size + 1)       ③
         for c in range(2, max_size + 1): ④
             n = 0
             for _ in range(total_classes):    ⑤
                 if shared_birthdays(c):
                     n = n + 1            ⑥
             p[c] = n / total_classes     ⑦
         return p
```

Find the minimize class size where the probability of a shared birthday > 50%

```
[12] # Cell 5
     prob = calc_probabilities()                   ⑧
     min_class_size = np.where(prob > 0.50)[0][0]  ⑨
     print(f"Min Class Size = {min_class_size}")

     Min Class Size = 23   ⑩
```

Run birthday_paradox.ipynb – Cell 6

Calculate the exact analytic probabilities for $2 \leq n \leq 80$ students using this formula: ①

$$p(n) \approx 1 - e^{-\frac{n^2}{730}}$$ ②

```
# Cell 6
n = np.arange(2, max_size + 1)      ③
p = 1.0 - np.exp(-(n**2) / 730)     ④
print(n)
print(p)
```

np.arrange() creates an array having numbers ***[start, stop)***

```
[ 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
 74 75 76 77 78 79 80]      ⑤
[0.00546447 0.01225308 0.02167936 0.0336668  0.04811883 0.06492009
 0.083938   0.10502447 0.12801783 0.15274488 0.17902307 0.20666274
 0.23546934 0.26524574 0.29579441 0.32691955 0.35842912 0.39013676
 0.42186346 0.45343913 0.48470395 0.51550954 0.5457198  0.57521171
 0.60387577 0.63161627 0.65835142 0.68401319 0.70854706 0.73191156
 0.75407772 0.77502834 0.79475721 0.81326825 0.83057458 0.84669754
 0.86166572 0.87551395 0.88828238 0.90001546 0.91076111 0.92056981
 0.92949381 0.93758643 0.94490135 0.95149204 0.95741122 0.96271041
 0.96743957 0.97164675 0.97537787 0.97867649 0.98158373 0.98413811
 0.98637557 0.98832943 0.99003044 0.99150683 0.99278438 0.99388653
 0.9948345  0.99564742 0.99634244 0.99693489 0.9974384  0.99786506
 0.99822552 0.99852915 0.99878417 0.99899772 0.99917602 0.99932446
 0.99944767 0.99954965 0.9996338  0.99970304 0.99975985 0.99980632
 0.99984423]      ⑥
```

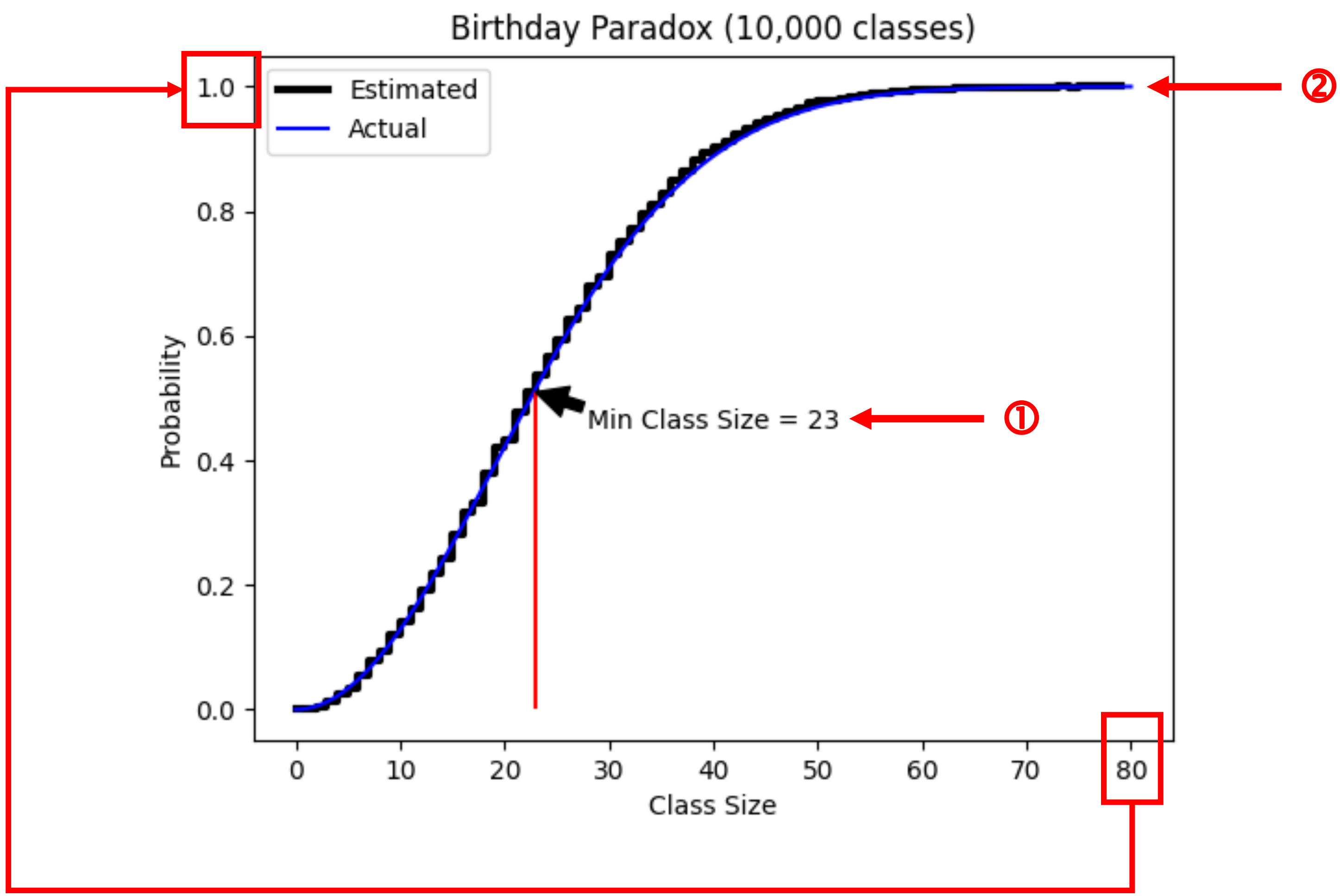
Apply the formula to every element in the array **n** and return each value in array **p**

Run birthday_paradox.ipynb – **Cell 7**

Graph both the discrete (estimated) and continuous (actual) probability curves ①

```
[7]  # Cell 7
     plt.step(range(max_size + 1), prob, color="black",       ②
              linewidth=3, label="Estimated")
     plt.plot(n, p, color="orange", label="Actual")           ③
     plt.title(f"Birthday Paradox ({total_classes:,} classes)")     ④
     plt.xlabel("Class Size")
     plt.ylabel("Probability")
     plt.vlines(min_class_size, 0, prob[min_class_size], color="blue")     ⑤
     plt.annotate(
         f"Min Class Size = {min_class_size}",
         xy=(min_class_size, prob[min_class_size]),
         xytext=(28, 0.45),     ⑥
         arrowprops={"facecolor": "black"},
     )
     plt.legend(loc="upper left")     ⑦
     plt.show()
```

Run birthday_paradox.py



Session **04** – Now You Know…

- How to identify **statements** and **scopes**
- How to define your own custom functions using **def()**
- How to create a **for()** loop to enumerate over a **range()**
- That **np.arange()** creates an array containing numbers in the closed-open interval **[start, stop)** using a given **step** size
- How to execute scopes based upon a condition using **if**
- How to make a scope loop using the **while** statement
- How to use **np.where()** function to select elements of an array that match a condition
- The **Numba** compiler is a **package** that can *significantly* accelerate your Python code

Task 04

- Update the code in **leibniz_formula.ipynb** to calculate the Leibniz series out to one million terms
- The Leibniz series is given by the sum:

$$s = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \underset{n=0}{1} - \overset{n=1}{\frac{1}{3}} + \underset{n=2}{\frac{1}{5}} - \overset{n=3}{\frac{1}{7}} + \underset{n=4}{\frac{1}{9}} - \overset{n=\infty}{\cdots}$$

- Don't use any **for()** loops – use **numpy arrays**!
- Calculate and display the value of $4 \times s$