notes

January 12, 2024

1 Theory

- **Identifiers:** Identifiers are names given to various program elements such as variables, functions, classes, etc. They are case-sensitive and must follow certain rules:
  - Must start with a letter (a-z, A-Z) or underscore (_)
  - Can be followed by letters, digits (0-9), or underscores
  - Cannot be a Python keyword (reserved word)
- **Keywords:** Keywords are predefined reserved words in Python that have special meanings and cannot be used as identifiers. For eg: `if`, `else`, `while`, `for`, `class`, `def`, `import`, `True`, `False`, `None`, etc
- **Statements:** A statement is a complete line of code that performs a specific action. Python programs consist of a sequence of statements. Common types of statements include assignment statements, conditional statements (if, elif, else), loop statements (for, while), and function calls
- **Expressions:** An expression is a combination of values, variables, and operators that can be evaluated to produce a result. Expressions can be simple, like a variable or a literal value, or complex, involving arithmetic, logical, or comparison operators
- **Variables:** Variables are used to store data values. They are identifiers associated with a memory location where data is stored. They must be declared (assigned a value) before they can be used

1.0.1 .py vs .pyc

Aspect	.py Files (Source Code)	.pyc Files (Compiled Bytecode)
File Type	Contains human-readable Python source code	Contains compiled bytecode (not readable)
Readability	Easily readable and editable by humans	Not intended for human readability or editing
Execution	Requires interpretation by the Python interpreter at runtime	Can be executed faster as it is pre-compiled to bytecode
Creation Process	Created by developers during coding	Automatically generated by the Python interpreter upon import
Extension	Has a `.py` extension	Has a `.pyc` extension
Size	Generally larger in size	Smaller in size compared to the corresponding `.py` file

Aspect	.py Files (Source Code)	.pyc Files (Compiled Bytecode)
Portability	Portable, can be shared and executed on any system with a Python interpreter	Platform-independent, can be shared but may have version dependencies
Editing	Editable using any text editor	Not meant for manual editing, changes should be made in the source `.py` file
Import Mechanism	Imported directly by the interpreter	If a `.pyc` file is present, it is imported instead of the source `.py` file
Reusability	Code needs to be interpreted each time it is run	Faster execution as the bytecode is stored and reused if the source is unchanged
Distribution	Source code may be distributed for others to read and modify	Bytecode can be distributed without revealing the source code
Interpretation Time	Longer interpretation time, as code is interpreted line by line	Shorter interpretation time, as bytecode is executed directly
Compilation Time	No explicit compilation step; interpreted directly	Compiled from source code during the first import or execution
Debugging	Source code facilitates easier debugging	Debugging may involve working with the source code rather than the bytecode
Example	`example.py`	`example.pyc`

1.0.2 Interactive mode vs script mode

Aspect	Interactive Mode	Script Mode
Mode of Operation	Executes code line by line in real-time	Executes an entire script or program
Input Method	Directly type commands into the terminal	Typically, write code in a text editor or IDE and save it as a script
Use Case	Useful for quick testing, experimentation, and learning	Ideal for developing larger, structured programs and automation tasks
User Interaction	Immediate feedback after each command	Limited interaction during execution, output is displayed at the end
Code Structure	Less focus on structure; encourages exploration and quick checks	Requires a well-structured script with functions, classes, and organization
Code Reusability	Limited reuse, as each line is executed individually	Promotes code reuse through functions, modules, and organized code
Debugging	Immediate error feedback after each line	Debugging may involve analyzing the entire script and using breakpoints
Execution Control	No direct control over the entire flow	Allows control flow using conditionals, loops, and functions
File Interaction	No direct file handling	Supports file handling and interaction with external files

Aspect	Interactive Mode	Script Mode
Environment Setup	Minimal setup required	Requires script creation, setup, and potential use of external modules
Execution Trigger	Starts automatically when the interpreter is launched	Requires explicit execution command (e.g., `python script.py`)

1.0.3 Features

1. **Easy to Learn and Read:** Python's syntax is straightforward and easy to understand, making it an excellent choice for beginners. It emphasizes code readability through the use of indentation rather than braces or keywords
2. **Extensive Standard Library:** Python comes with a comprehensive standard library that includes modules for various tasks like file handling, networking, regular expressions, and more. This extensive library reduces the need to write code from scratch and speeds up development
3. **Cross-Platform:** Python is available for a wide range of platforms, including Windows, macOS, Linux, and more. This cross-platform compatibility allows developers to write code once and run it on multiple operating systems
4. **Dynamic Typing:** Python uses dynamic typing, which means you don't need to declare the data type of a variable explicitly. This makes code more flexible and concise
5. **Interpreted Language:** Python is an interpreted language, which means you can run code directly without the need for compilation. This accelerates the development process and simplifies debugging
6. **High-Level Data Structures:** Python offers built-in support for high-level data structures like lists, dictionaries, and sets, making it easier to work with complex data
7. **Community Support:** Python has a large and active community of developers, which means you can find a wealth of resources, libraries, and frameworks to assist with your projects

1.0.4 Limitations

1. **Performance:** Python is not as fast as low-level languages like C or C++. It may not be the best choice for applications that require high performance and low latency, such as game development or system-level programming
2. **Global Interpreter Lock (GIL):** Python's Global Interpreter Lock can hinder multi-threaded performance, making it challenging to fully utilize multiple CPU cores for CPU-bound tasks
3. **Mobile App Development:** While Python has frameworks like Kivy and BeeWare for mobile app development, it is not as commonly used for this purpose as languages like Java or Swift
4. **Memory Consumption:** Python can be memory-intensive, which can be a limitation in resource-constrained environments

1.0.5 Applications

1. **Web Development:** Python is used to build web applications and websites using frameworks like Django and Flask

2. **Data Science and Machine Learning:** Python is the go-to language for data analysis, machine learning, and artificial intelligence, with libraries like NumPy, pandas, scikit-learn, and TensorFlow
3. **Scientific Computing:** Python is widely used in scientific research and engineering for tasks like simulations, data analysis, and visualization
4. **Automation and Scripting:** Python is excellent for automating repetitive tasks and writing scripts for tasks such as system administration and data processing
5. **Game Development:** Python is used for game development using libraries like Pygame
6. **Desktop Applications:** Python can be used to create desktop applications with graphical user interfaces (GUIs) using tools like PyQt and Tkinter
7. **Network Programming:** Python is commonly used for network-related tasks, including socket programming and building network servers
8. **DevOps and System Administration:** Python is used for tasks related to system administration, configuration management, and DevOps automation

1.0.6 Operators

1) **Arithmetic:** `+`, `-`, `*`, `/`, `//`, `%`, `**`
2) **Assignment:** `=`, `+=`, `-=`, `*=`, `/=`
3) **Comparison:** `==`, `!=`, `>`, `>=`, `<`, `<=`
4) **Logical:** `and`, `or`, `not`
5) **Membership:** `in`, `not in`
6) **Identity:** `is`, `is not`
7) **Bitwise:** `&`, `|`, `^`, `~`, `<<`, `>>`

1.0.7 Datatypes / Types of objects

1. **Numeric Types:**
   - Integers (`int`): Whole numbers like 5, -3, and 0
   - Floating-point numbers (`float`): Numbers with decimal points, such as 3.14 or -0.5
   - Complex numbers (`complex`): Numbers with a real and an imaginary part, like 2 + 3j
2. **Sequences:**
   - Lists (`list`): Ordered collections of items that can be of different data types
   - Tuples (`tuple`): Ordered, immutable collections of items
   - Strings (`str`): Sequences of characters
3. **Mappings:**
   - Dictionaries (`dict`): Collections of key-value pairs, where keys are unique
4. **Sets and Frozensets:**
   - Sets (`set`): Unordered collections of unique elements
   - Frozensets (`frozenset`): Immutable versions of sets
5. **Boolean Type:**
   - Booleans (`bool`): Represents `True` or `False` values
6. **None Type:**
   - `None`: Represents the absence of a value or a null value
7. **Custom Objects:**
   - You can create your own custom classes and objects, making Python an object-oriented language

1.0.8 Memory management

Memory management in Python is handled by the Python memory manager, which includes several components to ensure efficient memory usage:

1. **Reference Counting:** Python uses reference counting to keep track of how many references exist for each object. When an object's reference count drops to zero, it is automatically deleted (garbage collected)
2. **Garbage Collection:** Python also employs a garbage collector to identify and collect cyclic references, which reference counting alone cannot handle. The cyclic garbage collector detects and cleans up objects involved in circular references
3. **Memory Allocation:** Python's memory manager handles the allocation and deallocation of memory for objects dynamically. It ensures that memory is efficiently allocated and released as needed
4. **Memory Pools:** Python uses memory pools to manage memory allocation more efficiently. It divides memory into blocks of fixed sizes and allocates objects from these pools, reducing fragmentation and memory overhead
5. **Automatic Memory Management:** Python automatically manages memory for you, so you don't have to explicitly allocate or deallocate memory. This simplifies memory management and reduces the risk of memory leaks
6. **Caching and Reusing Objects:** Python caches and reuses certain immutable objects, like small integers and strings. This optimization reduces memory usage and improves performance

1.0.9 OOP-ness

1. **Objects and Classes:** Python allows you to define and use classes, which serve as blueprints for creating objects. Objects are instances of classes, and they encapsulate data and behavior
2. **Encapsulation:** OOP in Python promotes encapsulation, where data (attributes) and methods (functions) that operate on the data are grouped together within a class, providing a clear and organized structure
3. **Inheritance:** Python supports inheritance, allowing you to create new classes that inherit properties and behaviors from existing classes. Inheritance promotes code reuse and hierarchy
4. **Polymorphism:** Polymorphism is a key OOP concept in Python. It allows objects of different classes to be treated as objects of a common base class. This simplifies code and promotes flexibility
5. **Abstraction:** Abstraction in Python involves hiding complex implementation details and exposing a simplified interface. This simplification helps developers use classes and objects without needing to understand their internal workings
6. **Method Overriding:** Python allows you to override methods inherited from a base class in a derived class, providing flexibility and customization Here's a simple example of a Python class to illustrate the object-oriented nature of the language:

2 Comments

[1]:
```
# This is a comment
'''
This is also a comment
'''

print('Hello World')
```

```
Hello World
```

3 Multiple statements in single line

[2]:
```
a = 3; print(a)
```

```
3
```

4 Multiline statements

[3]:
```
a = 1 + 3 + 4 + \
    5 + 6 + 7
print(a)
```

```
26
```

5 Multiple variable assignmeents

[4]:
```
a,b = 4,5
print(a,b)
```

```
4 5
```

6 help() & __doc__

[5]:
```
def my_func():
    '''This is a doc-string'''
    pass

help(my_func)
print('-'*100)
print(my_func.__doc__)
```

```
Help on function my_func in module __main__:

my_func()
    This is a doc-string
```

```
--------------------------------------------------------------------------------
--------------------
This is a doc-string
```

7 print() and input()

[6]:
```
print?
```

```
Signature: print(*args, sep=' ', end='\n', file=None, flush=False)
Docstring:
Prints the values to a stream, or to sys.stdout by default.

sep
  string inserted between values, default a space.
end
  string appended after the last value, default a newline.
file
  a file-like object (stream); defaults to the current sys.stdout.
flush
  whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

[7]:
```
input?
```

```
Signature: input(prompt='')
Docstring:
Forward raw_input to frontends

Raises
------
StdinNotImplementedError if active frontend doesn't support stdin.
File:      /usr/local/lib/python3.11/site-packages/ipykernel/kernelbase.py
Type:      method
```

8 Types of statements

8.0.1 Empty Statements

- Empty statements are also known as "pass statements"
- They serve as a placeholder and do nothing when executed
- The `pass` keyword is used to create an empty statement
- Commonly used as a temporary placeholder when writing code or as a stub for future implementation

8.0.2 Simple Statements

- Simple statements are single-line statements that perform a specific action or operation
- They typically end with a newline character or a semicolon

- Common examples include assignment statements, function calls, and print statements

8.0.3 Compound Statements (Block Statements)

- Compound statements consist of one or more simple statements grouped together into a block
- They are often used to control program flow and define structures like loops and conditional statements
- Compound statements are defined using indentation (whitespace) in Python
- Common examples include if statements, while loops, for loops, and function definitions

[8]:
```
# Empty statement
def foo():
    pass  # Placeholder for function implementation


# Simple statements
x = 5  # Assignment statement
print("Hello, World!")  # Print statement
result = min(2, 3)  # Function call and assignment


# Compound statements
if x > 0:
    print("x is positive")  # Indented block as part of the if statement
else:
    print("x is not positive")  # Indented block as part of the else statement

while x < 5:
    print(x)  # Indented block as part of the while loop
    x += 1

def greet(name):
    print(f"Hello, {name}!")  # Indented block as part of the function␣
 ↪definition
```

```
Hello, World!
x is positive
```

9 Switch/Match

[9]:
```
x = 1

match x:
    case 1: print("x is 1")
    case 2: print("x is 2")
    case 3: print("x is 3")
    case _: print("x is something else")
```

```
x is 1
```

10 String fuctions

(https://www.w3schools.com/python/python_strings_methods.asp)

10.0.1 isprintable()

Basically if all characters are printable. Non-printable characters are things starting with `\` like `\n`, `\t`, or `\x07`

10.0.2 `isnumeric()` vs `isdigit()`

`isnumeric()` is more inclusive than `isdigit()` and can handle a wider range of numeric representations. For eg, `123½` will be returned `True` by `isnumeric()` but not by `isdigit()`

10.0.3 `find()` vs `index()`

`find()` returns `-1` if the substring wasn't found while `index()` raises a `ValueError`

10.0.4 `split()` vs `rsplit()`

- `split()` splits the string from left to right, while `rsplit()` splits it from right to left.
- By default, both methods return all substrings, but you can use the `maxsplit` parameter to limit the number of splits. When using `rsplit()`, the remaining part of the string appears as the first element in the result list.

10.0.5 `casefold()` vs `lower()`

- Use `lower()` when you need to perform simple case-insensitive operations and you are working with characters primarily from the Latin alphabet.
- Use `casefold()` when you need robust case-insensitive operations, especially for internationalization and localization, or when dealing with characters from various languages and scripts.

`casefold()` is generally recommended for case-insensitive string comparisons in most applications, as it provides more consistent and accurate results across a wide range of characters and languages.

10.0.6 `partition()` vs `split()`

`s.partition()` and `s.split()` are two string methods in Python used for splitting a string based on a specified delimiter. However, they work slightly differently:

1. `s.partition(delimiter)`:
    - The `s.partition(delimiter)` method splits the string `s` into three parts based on the first occurrence of the specified `delimiter`
    - It returns a tuple containing three elements: the part of the string before the delimiter, the delimiter itself, and the part of the string after the delimiter
    - If the delimiter is not found in the string, the method returns a tuple with the original string as the first element, followed by two empty strings

2. `s.split(delimiter, maxsplit)`:
    - The `s.split(delimiter, maxsplit)` method splits the string `s` into a list of substrings based on the specified `delimiter`
    - You can optionally specify the `maxsplit` parameter to control the maximum number of splits to perform
    - If the delimiter is not found in the string, the method returns a list with the entire string as a single element

- `partition()` always returns a tuple with three elements, whereas `split()` returns a list of substrings
- `partition()` splits the string only at the first occurrence of the delimiter, while `split()` can split the string at multiple occurrences
- `split()` provides more flexibility with the `maxsplit` parameter, allowing you to limit the number of splits

[10]:
```
x   = 'hello World 123!'
x_t = 'hello\tWorld\t123!'
x_l = 'hello\nWorld\n123!'

print('1 ', x.capitalize())                 # Capitalize the first character of␣
 ↪the string
print('2 ', x.casefold())                   # Perform case-folding on the string
print('3 ', x.center(20,'-'))               # Center-align the string within a␣
 ↪width of 20, padding with '-'
print('4 ', x.count('o'))                   # Count instances of substring 'o'
print('5 ', x.endswith('123!'))             # Check if the string ends with '123!'
print('6 ', x_t.expandtabs(10))             # Expand tab characters to spaces with␣
 ↪tab stops at 10
print('7 ', x.find('o'))                    # Find the first occurrence of 'o'
print('8 ', x.index('o'))                   # Get the index of the first␣
 ↪occurrence of 'o'
print('9 ', x.isalnum())                    # Check if all characters are␣
 ↪alphanumeric
print('10', x.isalpha())                    # Check if all characters are␣
 ↪alphabetic
print('11', x.isascii())                    # Check if the string is ASCII
print('12', x.isdecimal())                  # Check if all characters are decimal␣
 ↪digits
print('13', x.isdigit())                    # Check if all characters are digits
print('14', x.isidentifier())               # Check if the string is a valid␣
 ↪identifier
print('15', x.islower())                    # Check if all characters are lowercase
print('16', x.isnumeric())                  # Check if all characters are numeric
print('17', x.isprintable())                # Check if the string is printable
print('18', x.isspace())                    # Check if all characters are␣
 ↪whitespace
print('19', x.istitle())                    # Check if the string is titlecased
print('20', x.isupper())                # Check if all characters are uppercase
print('21', x.ljust(30,'-'))            # Left-justify the string within a␣
  ↪width of 30, padding with '-'
print('22', x.lower())                  # Convert the string to lowercase
print('23', x.lstrip('H'))              # Remove leading 'H' characters from␣
  ↪the string
print('24', x.replace('world','Earth'))  # Replace 'world' with 'Earth' in the␣
  ↪string
print('25', x.rfind('o'))               # Find the last occurrence of 'o'
print('26', x.rindex('o'))              # Get the index of the last occurrence␣
  ↪of 'o'
print('27', x.rjust(30, '-'))           # Right-justify the string within a␣
  ↪width of 30, padding with '-'
print('28', x.rsplit('o'))              # Split the string at 'o' from right␣
  ↪to left
print('29', x.rstrip('!'))              # Remove trailing '!' characters from␣
  ↪the string
print('30', x.split('o'))               # Split the string at 'o'
print('31', x_l.splitlines())           # Split the string into a list of lines
print('32', x.startswith('H'))          # Check if the string starts with 'H'
print('33', x.strip('!'))               # Remove leading and trailing '!'␣
  ↪characters from the string
print('34', x.swapcase())               # Swap the case of characters in the␣
  ↪string
print('35', x.title())                  # Convert the string to titlecase
print('36', x.upper())                  # Convert the string to uppercase
print('37', x.zfill(18))                # Pad the string with zeros to a total␣
  ↪width of 18
```

```
1  Hello world 123!
2  hello world 123!
3  --hello World 123!--
4  2
5  True
6  hello     World     123!
7  4
8  4
9  False
10 False
11 True
12 False
13 False
14 False
15 False
16 False
17 True
18 False
```

```
19 False
20 False
21 hello World 123!--------------
22 hello world 123!
23 hello World 123!
24 hello World 123!
25 7
26 7
27 --------------hello World 123!
28 ['hell', ' W', 'rld 123!']
29 hello World 123
30 ['hell', ' W', 'rld 123!']
31 ['hello', 'World', '123!']
32 False
33 hello World 123
34 HELLO wORLD 123!
35 Hello World 123!
36 HELLO WORLD 123!
37 00hello World 123!
```

11 Lists

[11]:
```python
x = [0, 1, 2, 3, 4, 4]

# Add an element to the end of an array
y = x.copy()
y.append(6)
print('1 ', y)

# Remove all elements from a list
y = x.copy()
y.clear()
print('2 ', y)

# Return the number of elements with the specified value
print('3 ', x.count(4))

# Add the elements of a list (or any iterable), to the end of the current list,␣
 ↪basically y + [8,9,1]
y = x.copy()
y.extend([8,9,1])
print('4 ', y)

# Return the index of the first element with the specified value
print('5 ', x.index(4))

# Add an element at the specified position (index, element)
```

```
y = x.copy()
y.insert(3,99)
print('6 ', y)

# Remove the element at the specified position
y = x.copy()
y.remove(3)
print('7 ', y)

# Remove the first item with the specified value
y = x.copy()
y.remove(4)
print('8 ', y)

# Reverse the order of the list, basically [::-1] but inplace
y = x.copy()
y.reverse()
print('9 ', y)

# Sort the list, basically sorted() but inplace
y = x.copy()
y.sort(reverse=True)
print('10', y)
```

```
1  [0, 1, 2, 3, 4, 4, 6]
2  []
3  2
4  [0, 1, 2, 3, 4, 4, 8, 9, 1]
5  4
6  [0, 1, 2, 99, 3, 4, 4]
7  [0, 1, 2, 4, 4]
8  [0, 1, 2, 3, 4]
9  [4, 4, 3, 2, 1, 0]
10 [4, 4, 3, 2, 1, 0]
```

12 Dictionaries

[12]:
```
x = {
    1: 'one',
    2: 'two',
    3: 'three'
}

# .clear() empties the dictionary
y = x.copy()
y.clear()
print('1 ', y)
```

```
# .fromkeys() makes a dictionary made via a list of keys. You can optionally␣
  ↪provide a value
y = dict.fromkeys([1,2,3], 'empty')
print('2 ', y)

# .pop deletes a key-value pair given a key, similar to del dict[key]. Returns␣
  ↪the deleted value. A default value can also be passed which would be␣
  ↪returned if key is not present. If keyis not present and no default value is␣
  ↪given, KeyError will arise
y  = x.copy()
z1 = y.pop(2)
z2 = y.pop(2, 'okay')
print('3 ', y, f'| {z1} | {z2}')

# .popitem() pops the most recently inserted item. Returns the deleted␣
  ↪key-value pair in a tuple. In versions before 3.7, it removes a random item
y = x.copy()
y[4] = 'four'
z = y.popitem()
print(3, y, f'| {z}')

# .setdeafult() behaves a bit, uniquely. A key and a an (optional) values is␣
  ↪passed (it is by default `None`). if the key exists, its value is returned.␣
  ↪Else the key is inserted with the passed value
y  = x.copy()
z1 = y.setdefault(3)
z2 = y.setdefault(4)
print(4, y, f'| {z1} | {z2}')

# .update() takes in a dict and basically pastes it over the original␣
  ↪dictionary. For every key in new disctionary, if the key exists in original␣
  ↪dict, its value (in the original dict) is updated. Else, the key-value pair␣
  ↪is inserted. Remember, the changes are made only in the original dict
y = x.copy()
y.update({3:'III', 4:'four'})
print(5, y)
```

```
1  {}
2  {1: 'empty', 2: 'empty', 3: 'empty'}
3  {1: 'one', 3: 'three'} | two | okay
3 {1: 'one', 2: 'two', 3: 'three'} | (4, 'four')
4 {1: 'one', 2: 'two', 3: 'three', 4: None} | three | None
5 {1: 'one', 2: 'two', 3: 'III', 4: 'four'}
```

13 Functions

- Max depth of recursion is 1000 by default (can be changed by using the `sys.setrecursionlimit()` function)

13.0.1 Base case vs recursive case

- Base case is the case where the function will not be called again, and it usually returns something
- Recursive case is the case which calls the function again

13.0.2 Direct vs indirect recursion

- In direct recursion, a function directly calls itself during its execution
- In indirect recursion, two or more functions are involved in a cycle of calling each other, creating a chain of function calls

```
def direct_recursion(n):
    if n <= 0:     return 1
    else:          return n * direct_recursion(n - 1)

def indirect_recursion_a(n):
    if n <= 0:     return 1
    else:          return n - indirect_recursion_b(n - 1)
def indirect_recursion_b(n):
    if n <= 0:     return 0
    else:          return n + indirect_recursion_a(n - 1)
```

13.0.3 Head vs tail recursion

- In head recursion, the recursive call is at the beginning of the function, before any other operations
- In tail recursion, the recursive call is the last operation in the function, and its result is immediately returned without further computation

```
def head_recursion(n):
    if n <= 0: return 1
    head_recursion(n - 1)
    n += 7
    print(n)

def tail_recursion(n, accumulator=1):
    if n <= 0: return accumulator
    n += 7
    return tail_recursion(n - 1, n * accumulator)
```

13.0.4 Types of arguments

- Positional (req.)
- Keyword (req.)

- Variable-length positional args (*args) (optional)
- Variable-length keyword args (**kwargs) (optional)

13.0.5 Conditions where recursions are used

[To fill in from PPT. If you are reading this, just fucking kill me]

13.0.6 Modules

- A module is a .py file containing Python definitions (functions, classes) and variables
- A module is a set of code or functions with the .py extension
- When we execute a program its module name is `__main__`
- This name is available in the variable `__name__`
- Two types of modules:
    - User defined modules
    - Standard library modules

13.0.7 Libraries

- A library is a collection of related modules or packages
- There can be multiple modules in a package

13.0.8 Packages

- A package is basically a directory with Python file and file with the extension as `__init.py__`
- Every directory residing inside the Python path which contains a file named `__init.py__` will be treated as a package by Python when the programmer can put multiple modules into a package
- A package can be imported like a normal module

14 Random module

[13]:
```
# https://docs.python.org/3/library/random.html
# https://www.w3schools.com/python/module_random.asp

import random
# Set seed. If the argument is omitted or None, the current system time is used.
 ↪ If randomness sources are provided by the operating system, they are used␣
 ↪instead of the system time
random.seed(100)

# Random float N such that 0 <= N <= 1
print('1 ', random.random())
# Random integer N such that a <= N <= b. Alias for randrange(a, b+1)
print('2 ', random.randint(1,100))
# Return a random floating point number N such that a <= N <= b for a <= b and␣
 ↪b <= N <= a for b < a
print('3 ', random.uniform(100, 200))
```

```
# Randomly selected element from range(start, stop, step). This is roughly␣
 ↪equivalent to choice(range(start, stop, step)) but supports arbitrarily␣
 ↪large ranges and is optimized for common cases. The positional argument␣
 ↪pattern matches the range() function. Keyword arguments should not be used␣
 ↪because they can be interpreted in unexpected ways. For example␣
 ↪randrange(start=100) is interpreted as randrange(0, 100, 1)
print('4 ', random.randrange(0,100,10))
# Return a random element from the non-empty sequence seq. If seq is empty,␣
 ↪raises IndexError
print('5 ', random.choice([1,2,3,4,5,6,7,8,9,10]))
# Shuffle the sequence x inplace
x = [1,2,3,4,5,6,7,8,9,10]
random.shuffle(x)
print('6 ', x)
# Return a k sized list of elements chosen from the population WITH replacement.
 ↪ If the population is empty, raises IndexError. If a weights sequence is␣
 ↪specified, selections are made according to the relative weights.␣
 ↪Alternatively, if a cum_weights sequence is given, the selections are made␣
 ↪according to the cumulative weights (perhaps computed using itertools.
 ↪accumulate()). For example, the relative weights [10, 5, 30, 5] are␣
 ↪equivalent to the cumulative weights [10, 15, 45, 50]. Internally, the␣
 ↪relative weights are converted to cumulative weights before making␣
 ↪selections, so supplying the cumulative weights saves work
print('7 ', random.choices([1,2,3,4,5,6,7,8,9,10], k=3))
# Return a k length list of unique elements chosen from the population sequence.
 ↪ Used for random sampling without replacement
print('8 ', random.sample([1,2,3,4,5,6,7,8,9,10], k=3))
```

```
1  0.1456692551041303
2  59
3  195.1717702826737
4  20
5  7
6  [3, 9, 8, 4, 10, 1, 5, 2, 7, 6]
7  [7, 7, 3]
8  [4, 5, 10]
```

15 Math module

[14]:
```
import math

# Return the floor of x, the largest integer less than or equal to x
print('1  ', math.floor(4.5))
# Return the ceiling of x, the smallest integer greater than or equal to x
print('2  ', math.ceil(4.5))
```

```
# Return x with the fractional part removed, leaving the integer part. This␣
↪rounds toward 0: trunc() is equivalent to floor() for positive x, and␣
↪equivalent to ceil() for negative x
print('3  ', math.trunc(4.5))
# Return n factorial as an integer. Raises ValueError if n is not integral or␣
↪is negative
print('4  ', math.factorial(6))
# Return the square root of x (as a float)
print('5  ', math.sqrt(9))
# Return the Euclidean distance between two points p and q, each given as a␣
↪sequence (or iterable) of coordinates. The two points must have the same␣
↪dimension. Returns a float
print('6  ', math.dist([0,0],[3,4]))
# Return the greatest common divisor of the specified integer arguments. If any␣
↪of the arguments is nonzero, then the returned value is the largest positive␣
↪integer that is a divisor of all arguments. If all arguments are zero, then␣
↪the returned value is 0. gcd() without arguments returns 0
print('7  ', math.gcd(2,4,0,6,10))
# Return the least common multiple of the specified integer arguments. If all␣
↪arguments are nonzero, then the returned value is the smallest positive␣
↪integer that is a multiple of all arguments. If any of the arguments is␣
↪zero, then the returned value is 0. lcm() without arguments returns 1
print('8  ', math.lcm(2,4,6,10))
# Return the product of all the elements in the input iterable. The default␣
↪start value for the product is 1. The start value is the value the product␣
↪is multiplied with
print('9  ', math.prod([1,7], start=0.5))
# (New in 3.12) Return the sum of products of values from two iterables p and q.
↪ Raises ValueError if the inputs do not have the same length
# print('10 ', math.sumprod([1,7], [4,5]))
# Return e raised to the power x, where e = 2.718281… is the base of natural␣
↪logarithms. This is usually more accurate than math.e ** x or pow(math.e, x)
print('11 ', math.exp(1))
# (New in 3.11) Return 2 raised to the power x. Returns a float
print('12 ', math.exp2(3))
# With one argument, return the natural logarithm of x (to base e). With two␣
↪arguments, return the logarithm of x to the given base, calculated as log(x)/
↪log(base)
print('13 ', math.log(math.e))
# Return the base-2 logarithm of x. This is usually more accurate than log(x, 2)
print('14 ', math.log2(2))
# Return the base-10 logarithm of x. This is usually more accurate than log(x,␣
↪10)
print('15 ', math.log10(10))
```

```
# Return x raised to the power y. pow(1.0, x) and pow(x, 0.0) always return 1.
 ↪0, even when x is a zero or a NaN. If both x and y are finite, x is␣
 ↪negative, and y is not an integer then pow(x, y) is undefined, and raises␣
 ↪ValueError. Unlike the built-in ** operator, math.pow() converts both its␣
 ↪arguments to type float. Use ** or the built-in pow() function for computing␣
 ↪exact integer powers
print('16 ', math.pow(9, 2))
# Convert angle x from radians to degrees. Returns a float
print('17 ', math.degrees(math.pi))
# Convert angle x from degrees to radians
print('18 ', math.radians(180))
# Return the sine of x radians
print('19 ', math.sin(0))
# Return the cosine of x radians
print('20 ', math.cos(0))
# Return the tangent of x radians
print('21 ', math.tan(0))

print('\nCONSTANTS\n')
print(' :    ', math.pi)
print('e:    ', math.e)
# A floating-point "not a number" (NaN) value. Equivalent to the output of␣
 ↪float('nan'). math.nan and float('nan') are not considered to equal to any␣
 ↪other numeric value, including themselves. To check whether a number is a␣
 ↪NaN, use the isnan() function to test for NaNs instead of is or ==
print('NAN: ', math.nan)
# A floating-point positive infinity. (For negative infinity, use -math.inf.)␣
 ↪Equivalent to the output of float('inf'):
print('∞:    ', math.inf)
```

```
1   4
2   5
3   4
4   720
5   3.0
6   5.0
7   2
8   60
9   3.5
11  2.718281828459045
12  8.0
13  1.0
14  1.0
15  1.0
16  81.0
17  180.0
18  3.141592653589793
```

```
19  0.0
20  1.0
21  0.0

CONSTANTS

 :    3.141592653589793
e:     2.718281828459045
NAN:   nan
∞:     inf
```

16 Matplotlib

16.0.1 Parameters of `.bar()` & `.hbar()`

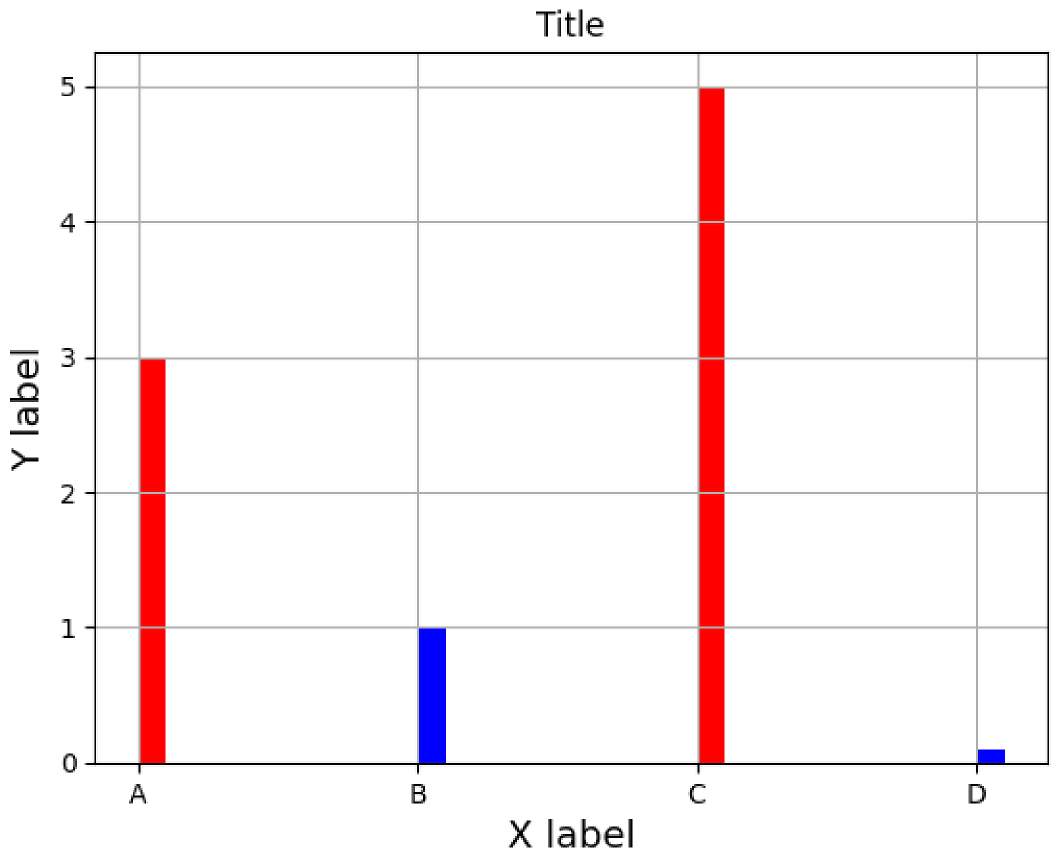
Parameter	Description
`x`	X labels. It must be hashable when using `.barh()`
`height`	The height(s) of the bars
`width`	The width(s) of the bars. By default is 0.8
`align`	Alignment of the labels w/ respect to the bars. Can be either `center` or `edge`. In `center` (default), the labels are in the center of the bar's bottom. In `edge`, the labels are on the left side
`color`	Can be a string or a list of strings (represing the color). If it is a single string, all bars will be of the same color. Else, the colors in the list will be provided (in the order they are in the list). If len(list) < len(height), then the cycle will start again

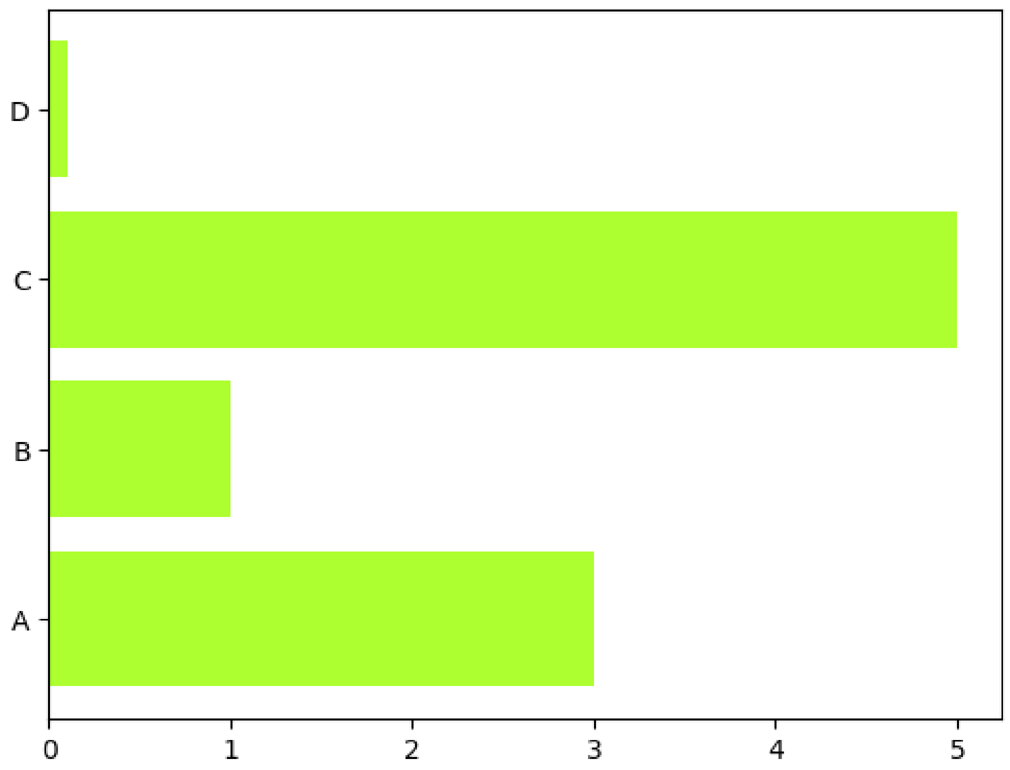
16.0.2 Parameters of `.pie()`

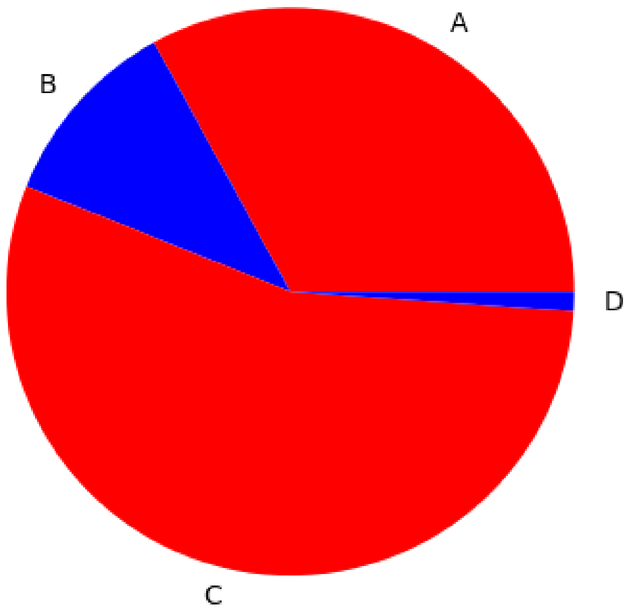
Parameter	Description
`x`	The wedge sizes
`labels`	A sequence of strings providing the labels for each wedge
`colors`	Same as `color` in `.bar()`

[19]:
```python
from matplotlib import pyplot as plt
data = {
    'A': 3,
    'B': 1,
    'C': 5,
    'D': 0.1
}
```

```
plt.title('Title')
plt.xlabel('X label', fontsize=14)
plt.ylabel('Y label', fontsize=14)
plt.grid(True)

plt.bar(data.keys(), data.values(), width=0.1, align='edge',␣
 ↪color=['red','blue'])
plt.show()

plt.barh(list(data.keys()), data.values(), color='greenyellow')
plt.show()

plt.pie(data.values(), labels=data.keys(), colors=['red','blue'])
plt.show()
```




D
C
B
A
0
1
2
3
4
5


A
B
D
C

17 Numpy

- Written partially in python, most of the parts requiring fast computation utilise C or C++
- It's the universal standard for working with numerical data in Python, and it's at the core of the scientific python and libraries like numpy, scikit-learn, and pandas
- It provides ndarray, a homogeneous n-dimensional array object, with methods to efficiently operate on it
- It can be used to perform a wide variety of mathematical operations on arrays
- It adds powerful data structures to Pthon that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices
- NumPy is a general-purpose array-processing package
- It provides a high-performance multidimensional array object, and tools for working with these arrays
- It contains various features including these important ones:
  - A powerful N-dimensional array object
  - Sophisticated (broadcasting) functions
  - Tools for integrating C/C++ and Fortran code

17.1 Why use numpy?

- Numpy aims to provide an array object that is up to 50x faster than traditional Python lists
- Numpy array consumes less memory than lists

17.2 Element wise operations (2 arrays)

(Cant be arsed with showing all of them via code)

These basically take in two arrays as arguments. Both should have the same shape. They are:

- `np.add()`
- `np.subtract()`
- `np.multiply()`
- `np.divide()`
- `np.remainder()`
- `np.power()` (First array elements raised to powers from second array)
- `np.maximum()`
- `np.minimum()`
- `np.logical_or()`
- `np.logical_and()`

17.3 Element wise operations (1 array)

These basically take in an array and return an array of the same shape. They are:

- `np.reciprocal()` (It makes a zero array for arrays with with the dtype being int)
- `np.sign()` (1 for positive, -1 for negative)
- `np.abs()`

- `np.floor()`
- `np.ceil()`
- `np.round()` (Can take another argument, ie the argument you would pass in built in function `round`)
- `np.sqrt()`

17.4 Parameters/arguments

17.4.1 `np.array`

- `dtype`: Data type that all the elements will be converted to. If a value in the list cannot be converted to the desired datatype, a `ValueError` is raised. It can be a python class (like `complex`, `float`, or `bool`), or can be a string representing the desired datatype like:
  - `int32` or `float32` (Here `32` represents the number of bytes the item takes)
  - `i4` or `f4` (Here `4` represents the number of bits the item takes)
  - `uint` (Unsigned integer)
  - `U` (Unicode string)
  - `S3` (String) (`3` represents the max number of characters in a element)

17.4.2 `np.zeros` and `np.ones`

- `size`: Must be a tuple, with first elem being no. of rows and second elem being no. of columns
- `dtype`

17.5 Differences

17.5.1 `.shape` vs `.size` vs `.itemsize`

- `.shape` returns a tuple containing the number of rows and number of columns of the array
- `.size` returns an integer ie the total number of elements in the array
- `.itemsize` returns the number of bytes occupied by a single elem

17.5.2 `min/max` vs `argmin/argmax`

- `min/max` returns the element
- `argmin/argmax` returns the index(s)

17.5.3 `.shape` vs `.reshape` vs `.resize`

(Note: I am talking about attributes/methods here. `np.resize` returns a copy and also starts repeating elements instead of using `0`. `np.reshape` works in the same way as the method)

Aspect	`.shape`	`.reshape`	`.resize`
Error	Throws an error if mxn != no. of elems	Throws an error if mxn != no. of elems	If mxn != no. of elems, sets the new elements to `0`

Aspect	`.shape`	`.reshape`	`.resize`
Inplace/returns	Inplace	Returns just a "view". So if changes to the original array is made, they are reflected	Inplace

17.5.4 `.ravel` vs `.flatten`

(Note: `.flat` is an attribute which represents the array in a 1D view) (Also note: there is also a function called `np.ravel` which works in the same way as the `ravel` method)

- They both return shit, neither is inplace
- `.ravel` returns a "view", while `.flatten` returns a copy
- `.ravel` is faster as it does not occupy any memory

17.6 Other shit

- `numpy.ndarray` and `numpy.array` are essentially the same thing. The `numpy.array` function is a constructor for creating a new array, and the result is an instance of the `numpy.ndarray` class. So, when you use `numpy.array`, you are creating a NumPy array object, which is an instance of the `numpy.ndarray` class
- axis 0 means column, axis 1 means row
- `any`, `all`, and `median` work in the same syntax as `min` or `max`
- Like pandas dataframes, doing something like `2*arr` will double all the elems in `arr`
- When doing comparison operators between two arrays, it must be ensured that both arrays have the same shape. The resultant would be an array of the same shape with each element being `True` or `False` depending on the corresponding elements in the original arrays

[15]:
```python
import numpy as np

arr = np.array([[1,2,3,4,5,6]], dtype='f4')
print('1 ', type(arr))
print('2 ', arr[0,1])
print('3 ', arr)
print('4 ', arr.astype(int))
print('5 ', arr.dtype)
print('6 ', arr.size)
print('7 ', arr.shape)
print('8 ', arr.ndim)
print('9 ', arr.itemsize)
print('10', arr.T)
print('11', np.empty((2,3), float))
print('12', np.zeros((2,3), bool))
print('13', np.ones((1,4), int))
print('14', np.eye(2, dtype=int)) # Create a (square) identity matrix of n rows␣
 ↪and n cols
print('15', np.random.random( (1,3) )) # Create a 1x3 array with random values
print('16', np.arange(0,10,2)) # Basically np.array( range(0,10,2)
print('17', np.linspace(4,100,25)) # Create an array with 25 elems, with the␣
  ↪first elem being 4 and the last elem being 25. Gaps b/w the elems is equal
arr2 = np.array([[1,2,3,4,5,6]], dtype='i')
print('20', arr2)
print('19', np.array_equal(arr, arr2))

print('\n'+'-'*100+'\n')

print('SHAPE, RESHAPE, & RESIZE\n')
arr = np.array([[1,2,3,4,5,6]], dtype='f4')
arr2 = arr.copy()
print('20', arr2)
arr2.shape=(2,3)
print('21', arr2)
print('22', arr.reshape(2,3))
arr2 = arr.copy()
arr2.resize(3,3)
print('23', arr2)

print('\n'+'-'*100+'\n')

print('SUM & CUMSUM\n')
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('24', arr)
print('25', arr.sum()) # Returns an int ie sum of all vals
print('26', arr.sum(axis=0)) # Returns a 1D arr that contains sums of all cols
print('27', np.sum(arr,axis=1)) # Returns a 1D arr that contains sums of all␣
  ↪rows
print('28', arr.cumsum()) # Returns a 1D arr
print('29', arr.cumsum(axis=0)) # Returns a 2D arr
print('30', np.cumsum(arr,axis=1)) # Returns a 2D arr

print('\n'+'-'*100+'\n')

print('PROD & CUMPROD\n')
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('31', arr)
print('32', arr.prod()) # Returns an int ie product of all vals
print('33', arr.prod(axis=0)) # Returns a 1D arr that contains product of all␣
  ↪cols
print('34', np.prod(arr,axis=1)) # Returns a 1D arr that contains product of␣
  ↪all rows
print('35', arr.cumprod()) # Returns a 1D arr
print('36', arr.cumprod(axis=0)) # Returns a 2D arr
print('37', np.cumprod(arr,axis=1)) # Returns a 2D arr
```

```python
print('\n'+'-'*100+'\n')

print('MEAN\n')
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('38', arr)
print('39', arr.mean()) # Returns an int ie mean of all vals
print('40', arr.mean(axis=0)) # Returns a 1D arr that contains means for every␣
 ↪col
print('41', np.mean(arr, axis=1)) # Returns a 1D arr that contains means for␣
 ↪every row

print('\n'+'-'*100+'\n')

print('MIN & MAX\n')
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('42', arr)
print('43', arr.min())
print('44', arr.min(axis=0))
print('45', np.min(arr, axis=1))
print('46', arr.max())
print('47', arr.max(axis=0))
print('48', np.max(arr, axis=1))
```

```
1  <class 'numpy.ndarray'>
2  2.0
3  [[1. 2. 3. 4. 5. 6.]]
4  [[1 2 3 4 5 6]]
5  float32
6  6
7  (1, 6)
8  2
9  4
10 [[1.]
 [2.]
 [3.]
 [4.]
 [5.]
 [6.]]
11 [[4.9e-324 9.9e-324 1.5e-323]
 [2.0e-323 2.5e-323 3.0e-323]]
12 [[False False False]
 [False False False]]
13 [[1 1 1 1]]
14 [[1 0]
 [0 1]]
15 [[0.50860435 0.43804196 0.50323485]]
16 [0 2 4 6 8]
```

```
17 [  4.   8.  12.  16.  20.  24.  28.  32.  36.  40.  44.  48.  52.  56.
  60.  64.  68.  72.  76.  80.  84.  88.  92.  96. 100.]
20 [[1 2 3 4 5 6]]
19 True

--------------------------------------------------------------------------------
--------------------

SHAPE, RESHAPE, & RESIZE

20 [[1. 2. 3. 4. 5. 6.]]
21 [[1. 2. 3.]
 [4. 5. 6.]]
22 [[1. 2. 3.]
 [4. 5. 6.]]
23 [[1. 2. 3.]
 [4. 5. 6.]
 [0. 0. 0.]]

--------------------------------------------------------------------------------
--------------------

SUM & CUMSUM

24 [[1 2 3]
 [4 5 6]
 [7 8 9]]
25 45
26 [12 15 18]
27 [ 6 15 24]
28 [ 1  3  6 10 15 21 28 36 45]
29 [[ 1  2  3]
 [ 5  7  9]
 [12 15 18]]
30 [[ 1  3  6]
 [ 4  9 15]
 [ 7 15 24]]

--------------------------------------------------------------------------------
--------------------

PROD & CUMPROD

31 [[1 2 3]
 [4 5 6]
 [7 8 9]]
32 362880
33 [ 28  80 162]
```

```
34 [  6 120 504]
35 [     1      2      6     24    120    720   5040  40320 362880]
36 [[  1   2   3]
 [  4  10  18]
 [ 28  80 162]]
37 [[  1   2   6]
 [  4  20 120]
 [  7  56 504]]

--------------------------------------------------------------------------------
--------------------

MEAN

38 [[1 2 3]
 [4 5 6]
 [7 8 9]]
39 5.0
40 [4. 5. 6.]
41 [2. 5. 8.]

--------------------------------------------------------------------------------
--------------------

MIN & MAX

42 [[1 2 3]
 [4 5 6]
 [7 8 9]]
43 1
44 [1 2 3]
45 [1 4 7]
46 9
47 [7 8 9]
48 [3 6 9]
```

18 File handling

- The `.tell()` method returns an integer representing where the pointer currently is
- In `.seek(cookie:int, whence:int=0)`, `cookie` represents the offset and `whence` represents "from where"

18.0.1 File access modes

Mode	Description
`r rb`	Read. If file does not exist, an I/O error is raised
`w rb`	Write

Mode	Description
`a ab`	Append
`r+ rb+ r+b`	Reading and writing both allowed. If file does not exist, an I/O error is raised
`w+ wb+ w+b`	Reading and writing both allowed. If file exists, it is truncated. If file does not exist, it is created
`a+ ab+ a+b`	Reading and writing both allowed. Exactly like `w+`, but the pointer is placed at the end of the file

18.0.2 `.flush()`

- When we write any data to file, python hold everything in buffer (temporary memory) and pushes it onto actual file later. If you want to force Python to write the content of buffer onto storage, you can use the `.flush()` method
- Python automatically flushes the files when closing them i.e. it will be implicitly called by the close(), BUT if you want to flush before closing any file you can use `.flush()`

18.0.3 `whence` values

Value	Meaning
0	Beginning of the file
1	Position of file pointer
2	End of the file

[8]:
```python
from csv import reader, writer

fields = ['Letter', 'Word']
data = [
    ['A', 'Alpha'  ],
    ['B', 'Beta'   ],
    ['C', 'Charlie'],
    ['D', 'Delta'  ]
]

with open('data.csv', 'w') as f:
    csv_w = writer(f)
    csv_w.writerow(fields)
    csv_w.writerows(data)

with open('data.csv', 'r') as f:
    csv_r = reader(f)
    for row in csv_r:
        print(row)
```

```
['Letter', 'Word']
['A', 'Alpha']
```

```
['B', 'Beta']
['C', 'Charlie']
['D', 'Delta']
```