Biopython

Ali Özgür Argunşah - Göker Arpağ

Molecular Biology and Genetics
Kadir Has University

January 15, 2025

What is Python?

- Python is a very versatile programming language.
- Created by Guido van Rossum in 1991.

- Widely used for various purposes:
  - Web development (server-side).
  - Software development.
  - Mathematics.
  - System scripting.

What is Biopython?

Officially – The Biopython Project is an international association of developers of freely available Python tools for computational molecular biology.

Simple terms – Develop reusable Python code for anything related to biology and computing.

Practical Things

- Established in 1999
- `http://www.biopython.org`
- Members of the Open Bioinformatics Foundation
- Active mailing lists; semi-regular releases; CVS. . . everything you like in an open-source project

What Can Python Do?

- Python's capabilities:
  - Create web applications on servers.
  - Integrate with software to build workflows.
  - Connect to databases and manipulate files.
  - Handle big data and perform complex mathematics.
  - Machine/Deep Learning.
  - Rapid prototyping and production-ready software development.

What does Biopython have for you?

Sequence-type Data – Sequences, features on sequences, storage in SQL databases (BioSQL), retrieval from on-line databases (NCBI)

Biological File Formats – FASTA, GenBank

Manipulating Sequences – Transcription, translation, slicing sequences

Bioinformatics programs – Dealing with BLAST, Clustalw, EMBOSS

Structure Data – PDB parsing, representation and analysis

Microarray Data – Clustering data, reading and writing

Substitution Matrices – Creating, manipulating, common matrices

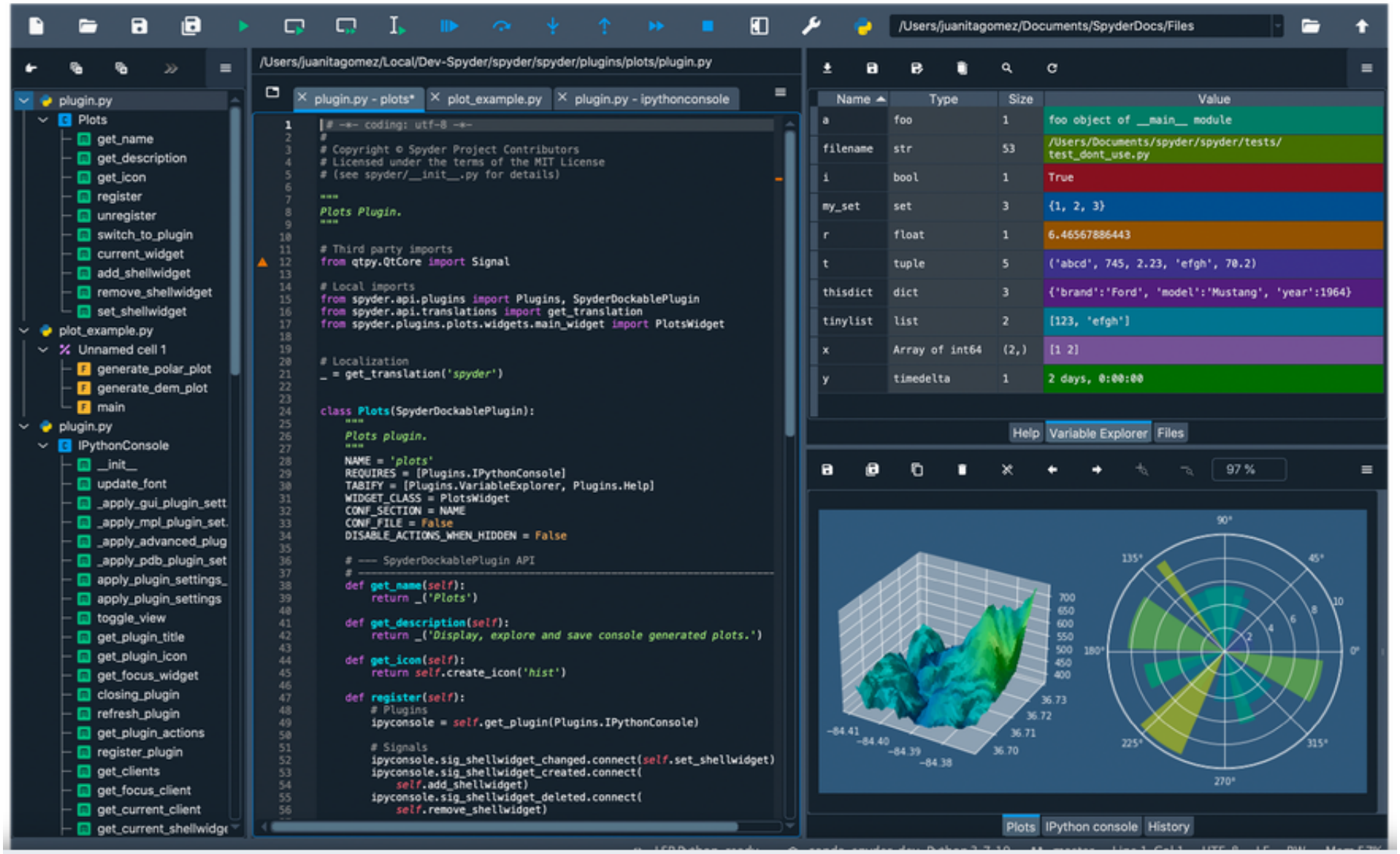
Why Python?

- Advantages of Python:
  - Cross-platform compatibility (Windows, Mac, Linux, Raspberry Pi, etc.).
  - Simple, English-like syntax.
  - Concise code with fewer lines.
  - Interpreted language for quick prototyping.
  - Supports procedural, object-oriented, and functional programming.

Good to Know

- Python 3 is the most recent major version.
- Python 2 is still in use but not updated, mainly for security.
- In this class, we'll use Python 3.
- Python can be written in a text editor or IDEs like Spyder, PyCharm, etc.

i.e. Spyder


/Users/juanitagomez/Documents/SpyderDocs/Files
/Users/juanitagomez/Local/Dev-Spyder/spyder/spyder/plugins/plots/plugin.py
plugin.py - plots*
plot_example.py
plugin.py - ipythonconsole
Name
Type
Size
Value
Help
Variable Explorer
Files
97 %
Plots
IPython console
History

Python Syntax

- Python emphasizes readability.
- Syntax resembles English with mathematical influences.
- Uses new lines for commands, not semicolons or parentheses.
- Relies on indentation for defining scope (loops, functions, classes).

Python Installation

- To check if you have Python installed:

  On Windows – search for Python in the start bar or run `python --version` in Command Line (cmd.exe).

  On Linux or Mac – open the terminal and type `python --version`.
- If Python is not installed, download it from `https://www.python.org/`.

Python Quickstart

- Python is an interpreted programming language.
- Write Python (.py) files in a text editor and execute them using the Python interpreter.

Running Python Files

- To run a Python file from the command line:
  - On Windows: `python filename.py`.
  - Replace `filename.py` with the name of your Python file.

Example: helloworld.py

helloworld.py

```
print("Hello, World!")
```

- Save the file, navigate to its directory in the command line, and run:
- `python helloworld.py`
- Output: "Hello, World!"

Python Command Line

- Python can be run as a command line.
- On Windows, Mac, or Linux, type `python` or `py` on the command line.
- You can write and execute Python code interactively.

Example: Python Command Line

```
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0]
on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello, World!")
Hello, World!
```

Exiting the Python Command Line

- To exit the Python command line interface, simply type:
- `exit()`

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- In Python, indentation is essential to indicate a block of code.

Example: Indentation in Python

```
# Python program showing
# indentation
site = 'nerd'
if site == 'nerd':
    print('You are a nerd...')
else:
    print('You are not a nerd.')
print('Nice to meet you!')
```

- Python uses indentation to define code blocks.
- Skipping indentation leads to a syntax error.

Example: Incorrect Indentation

```
# Python program showing
# indentation
# Syntax Error
site = 'nerd'
if site == 'nerd':
print('You are a nerd...')
else:
print('You are not a nerd.')
print('Nice to meet you!')
```

- Incorrect indentation causes syntax errors.
- Consistent indentation is crucial.

Example: Different Indentation

```
# Syntax Error
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

- Using different indentation levels in the same block leads to syntax errors.
- Note: Python uses 4 spaces as indentation by default. However, the number of spaces is up to you, but a minimum of 1 space has to be used.

Python Variables

- Variables in Python are created when assigned a value.
- No need to declare variables explicitly.

Example: Variables in Python

```
# Variables in Python
x = 5
y = "Hello, World!"
```

Comments in Python

- Python supports comments for in-code documentation.
- Comments start with # and are not executed.

Example: Comments in Python

```
# This is a comment.
print("Hello, World!")
```

Comments in Python

- Comments can be used to explain Python code.
- They enhance code readability.
- Comments can prevent code execution when testing.

Creating a Comment

```
# This is a comment
print("Hello, World!")
```

- Comments begin with #, and Python ignores them.

End-of-Line Comments

```
print("Hello, World!") # This is a comment
```

- Comments can be placed at the end of a line, and Python ignores the rest of the line.

Preventing Execution

```
# print("Hello, World!")
print("Cheers, Mate!")
```

- Comments can be used to prevent code execution.

Multi-line Comments

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

- Python doesn't have a specific multiline comment syntax.
- You can use triple-quoted strings to create multiline comments.

Variables in Python

- Variables are containers for storing data values.

Creating Variables

```
x = 5
y = "John"
print(x)
print(y)
```

- Variables are created when values are assigned.
- Python doesn't require declaring a variable's type.

Changing Variable Type

```
x = 4       # x is an int
x = "Sally" # x is now a str
print(x)
```

- Variables can change type after being set.

Casting

```
x = str(3)    # x is a str '3'
y = int(3)    # y is an int 3
z = float(3)  # z is a float 3.0
```

- You can specify the data type of a variable with casting.

Get the Type

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

- The ‘type()‘ function gets the data type of a variable.

String Quotes

```
x = "John"
# is the same as
x = 'John'
```

- String variables can use single or double quotes interchangeably.

Case Sensitivity

```
a = 4
A = "Sally"
# A will not overwrite a
```

- Variable names are case-sensitive in Python.

Variable Names in Python

- Variable names can be short (e.g., x, y) or descriptive (e.g., age, carname, total_volume).
- Rules for Python variables:
  - Must start with a letter or underscore (_).
  - Cannot start with a number.
  - Can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
  - Are case-sensitive (age, Age, and AGE are different variables).
  - Cannot be any of the Python keywords.

Example: Legal Variable Names

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Example: Illegal Variable Names

```
# These are illegal variable names
2myvar = "John"
my-var = "John"
my var = "John"
```

Remember: Case Sensitivity

- Variable names are case-sensitive in Python.

Multi-Word Variable Names

- Variable names with multiple words can be challenging to read.
- Techniques for making them more readable:

  Camel Case: Each word (except the first) starts with a capital letter (e.g., myVariableName).

  Pascal Case: Each word starts with a capital letter (e.g., MyVariableName).

  Snake Case: Words are separated by underscore characters (e.g., my_variable_name).

Multiple Values to Multiple Variables

- Python allows you to assign values to multiple variables in one line.

Example: Multiple Values to Multiple Variables

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

- Ensure the number of variables matches the number of values to avoid errors.

One Value to Multiple Variables

- You can assign the same value to multiple variables in one line.

Example: One Value to Multiple Variables

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Unpacking a Collection

- If you have a collection of values in a list, tuple, etc., Python allows you to extract the values into variables. This is called unpacking.

Example: Unpacking a List

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Outputting Variables in Python

- The Python ‘print()‘ function is often used to output variables.

Example: Outputting a String Variable

```
x = "Python is awesome"
print(x)
```

Example: Outputting Multiple Variables with ‘print()‘

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

Example: Outputting Multiple Variables with ‘+‘ Operator

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

Outputting Numbers with ‘print()‘

```
x = 5
y = 10
print(x + y)
```

Example: Mixing String and Number with ‘print()‘

```
x = 5
y = "John"
print(x + y)
```

- Attempting to combine a string and a number with ‘+‘ will result in an error.

Best Practice: Separating Variables with Commas

```
x = 5
y = "John"
print(x, y)
```

- The recommended way to output multiple variables in ‘print()‘ is to separate them with commas, supporting different data types.

Built-in Data Types in Python

- In programming, data types are essential.
- Python has several built-in data types categorized into:

1. Text Type: ‘str‘
2. Numeric Types: ‘int‘, ‘float‘, ‘complex‘
3. Sequence Types: ‘list‘, ‘tuple‘, ‘range‘
4. Mapping Type: ‘dict‘
5. Set Types: ‘set‘, ‘frozenset‘
6. Boolean Type: ‘bool‘
7. Binary Types: ‘bytes‘, ‘bytearray‘, ‘memoryview‘
8. None Type: ‘NoneType‘

Getting the Data Type

```
x = 5
print(type(x))
```

- You can use the ‘type()‘ function to determine the data type of an object.

Setting the Data Type in Python

- In Python, the data type is automatically set when you assign a value to a variable.

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = "name" : "John", "age" : 36	dict
x = "apple", "banana", "cherry"	set
x = frozenset("apple", "banana", "cherry")	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview
x = None	NoneType

Setting the Specific Data Type

- You can specify the data type explicitly using constructor functions.

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Python Numbers

- There are three numeric types in Python:
  1. int
  2. float
  3. complex
- Variables of numeric types are created when you assign a value to them.

Example: Numeric Variables

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

Verifying the Data Type

```
print(type(x))
print(type(y))
print(type(z))
```

Integers (int)

```
x = 1
y = 35656222554887711
z = -3255522
```

Floats (float)

```
x = 1.10
y = 1.0
z = -35.59
```

Floats with Scientific Notation

```
x = 35e3
y = 12E4
z = -87.7e100
```

Complex Numbers (complex)

```
x = 3+5j
y = 5j
z = -5j
```

Type Conversion

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
a = float(x)      # convert from int to float
b = int(y)        # convert from float to int
c = complex(x)    # convert from int to complex
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

Random Numbers in Python

```
import random
print(random.randrange(1, 10))
```

Specify a Variable Type in Python

- There may be times when you want to specify a type for a variable. This can be done with casting.
- Python is an object-oriented language and uses constructor functions for casting to specific data types.

Casting to Integers

```
# Integers:
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Casting to Floats

```
# Floats:
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

Casting to Strings

```
# Strings:
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

Strings in Python

- Strings in Python are surrounded by either single quotation marks or double quotation marks. ‘’hello’‘ is the same as ‘”hello”‘.
- You can display a string literal with the ‘print()‘ function:

```
print("Hello")
print('Hello')
```

Assign String to a Variable

```
a = "Hello"
print(a)
```

Multiline Strings

```
# Using three double quotes:
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
# Or three single quotes:
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Strings are Arrays

```
# Get the character at position 1
a = "Hello, World!"
print(a[1])
```

Looping Through a String

```
# Loop through the letters in the word "banana"
for x in "banana":
  print(x)
```

String Length

```
# Get the length of a string
a = "Hello, World!"
print(len(a))
```

Check String

```
# Check if "free" is present in the text
txt = "The best things in life are free!"
print("free" in txt)
```

Check String (contd.)

```
# Print only if "free" is present
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

Check String (NOT in)

```
# Check if "expensive" is NOT present in the text
txt = "The best things in life are free!"
print("expensive" not in txt)
```

Check String (NOT in) (contd.)

```
# Print only if "expensive" is NOT present
txt = "The best things in life are free!"
if "expensive" not in txt:
  print("No, 'expensive' is NOT present.")
```

Slicing Strings in Python

- You can return a range of characters from a string using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

Slicing Example

```
# Get the characters from position 2 to 4 (not included)
b = "Hello, World!"
print(b[2:5])
```

Slice From the Start

```
# Get the characters from the start
to position 4 (not included)
b = "Hello, World!"
print(b[:5])
```

Slice To the End

```
# Get the characters from position 2 to the end
b = "Hello, World!"
print(b[2:])
```

Negative Indexing

```
# Get the characters from position -5 to -2
b = "Hello, World!"
print(b[-5:-2])
```

String Methods in Python

Upper Case

The `upper()` method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Output: "HELLO, WORLD!"

Lower Case

The `lower()` method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Output: "hello, world!"

String Methods in Python

Remove Whitespace

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip())  # returns "Hello, World!"
```

String Methods in Python

Replace String

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Output: "Jello, World!"

String Methods in Python

Split String

The `split()` method returns a list where the text between the specified separator becomes the list items:

```
a = "Hello, World!"
print(a.split(","))
```

Output: ['Hello', ' World!']

String Concatenation in Python

String Concatenation

To concatenate, or combine, two strings, you can use the + operator.

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Output: "HelloWorld"

String Concatenation in Python

Adding a Space

To add a space between concatenated strings, include a space within double quotes:

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

Output: "Hello World"

String Formatting in Python

String Formatting

In Python, you can combine strings and numbers using the `format()` method.

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders {} are.

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Output: "My name is John, and I am 36"

You can use index numbers {0}, {1}, etc., to specify the order of arguments.

Common String Methods in Python

- `capitalize()`: Convert the first character to upper case.
- `casefold()`: Convert the string into lower case.
- `center()`: Return a centered string.
- `count()`: Return the number of times a specified value occurs in a string.
- `encode()`: Return an encoded version of the string.
- `endswith()`: Return `True` if the string ends with the specified value.
- `expandtabs()`: Set the tab size of the string.
- `find()`: Search the string for a specified value and return its position.

Booleans in Python

- Booleans represent one of two values: `True` or `False`.
- In Python, you often need to evaluate expressions to determine if they are `True` or `False`.
- When comparing values, Python returns either `True` or `False`.

Examples

- `10 > 9` is `True`
- `10 == 9` is `False`
- `10 < 9` is `False`

Most Values are True

- Almost any value is evaluated as `True` if it has content.
- Strings, numbers (except 0), lists, tuples, sets, and dictionaries (if not empty) are evaluated as `True`.

Booleans in Python cont.

Some Values are False

- Empty values like `()`, `[]`, {}, `""`, and `0` are evaluated as `False`.
- The value `False` is also `False`.

Python Operators

- Operators are used to perform operations on variables and values.
- Python divides operators into several groups:
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators

Operator Precedence

Operator precedence defines the order in which operations are performed. Parentheses have the highest precedence.

- Highest Precedence: `()`
- Lowest Precedence: `or`

Python Operators cont.

Arithmetic Operators

- + Addition
- - Subtraction
- * Multiplication
- / Division
- % Modulus
- ** Exponentiation
- // Floor division

Python Lists

- Lists are used to store multiple items in a single variable.
- Lists are ordered, changeable, and allow duplicate values.
- List items are indexed.

List Creation

Lists are created using square brackets.

```
thislist = ["apple", "banana"]
```

List Properties

- Ordered: Items have a defined order.
- Changeable: You can add, change, and remove items.
- Allow Duplicates: Items can have the same value.
- Length: Use `len()` to find the number of items.
- Data Types: Lists can contain various data types.

Python Lists cont.

Examples

- List with strings, integers, and boolean values:

```
mylist = ["abc", 34, True, 40, "male"]
```

- What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))  # Output: <class 'list'>
```

Accessing Items in Lists

List Indexing

- List items are indexed, starting from 0.
- You can access items by referring to their index number.

Example

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])  # Output: "banana"
```

Accessing Items in Lists contd.

Negative Indexing

- Negative indexing starts from the end of the list.
- -1 refers to the last item, -2 to the second last, and so on.

Example

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])  # Output: "cherry"
```

Accessing Items in Lists contd.

Range of Indexes

- You can specify a range of indexes to get a subset of the list.
- The range includes the start index but excludes the end index.

Example

```
thislist = ["apple","banana","cherry","orange","kiwi","melon"]
print(thislist[2:5])
# Output: ["cherry","orange","kiwi"]
```

Example

```
thislist = ["apple","banana","cherry","orange","kiwi","melon"]
print(thislist[:4])
# Output: ["apple","banana","cherry","orange"]
```

Accessing Items in Lists (Contd.)

Range of Negative Indexes

- You can specify negative indexes to start the search from the end of the list.

Example

```
thislist = ["apple","banana","cherry","orange","kiwi","melon"]
print(thislist[-4:-1])
# Output: ["orange","kiwi","melon"]
```

Accessing Items in Lists (Contd.)

Check If Item Exists

- To check if a specific item is in a list, use the `in` keyword.

Example

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Appending and Inserting Items in Lists

Appending Items

- To add an item to the end of a list, use the `append()` method.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
# Output: ["apple", "banana", "cherry", "orange"]
```

Appending and Inserting Items in Lists contd.

Inserting Items

- To insert an item at a specified index, use the `insert()` method.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
# Output: ["apple", "orange", "banana", "cherry"]
```

Appending and Inserting Items in Lists contd.

Extending Lists

- To append elements from another list or iterable to the current list, use the `extend()` method.

Example (Python)

```
thislist = ["apple","banana","cherry"]
tropical = ["mango","pineapple","papaya"]
thislist.extend(tropical)
print(thislist)
# Output: ["apple","banana","cherry","mango","pineapple","papaya"]
```

Looping Through a List

Using a For Loop

- You can loop through list items using a for loop.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
# Output: "apple", "banana", "cherry"
```

Looping Through a List contd.

Looping Through Index Numbers

- You can loop through list items by referring to their index number.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
# Output: "apple", "banana", "cherry"
```

Looping Through a List contd.

Using a While Loop

- You can loop through list items using a while loop.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
# Output: "apple", "banana", "cherry"
```

Looping Through a List contd.

List Comprehension

- List Comprehension offers a concise way to loop through lists.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
# Output: "apple", "banana", "cherry"
```

Looping Through a List

Using a For Loop

- You can loop through list items using a for loop.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
# Output: "apple", "banana", "cherry"
```

Looping Through a List

Looping Through Index Numbers

- You can loop through list items by referring to their index number.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
# Output: "apple", "banana", "cherry"
```

Looping Through a List

Using a While Loop

- You can loop through list items using a while loop.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
# Output: "apple", "banana", "cherry"
```

Sorting a List

Alphanumerical Sorting

- Lists can be sorted alphanumerically using the `sort()` method.

Example (Python)

```
thislist = ["orange","mango","kiwi","pineapple","banana"]
thislist.sort()
print(thislist)
# Output: ["banana","kiwi","mango","orange","pineapple"]
```

Sorting a List contd.

Numerical Sorting

- Lists can also be sorted numerically using the `sort()` method.

Example (Python)

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
# Output: [23, 50, 65, 82, 100]
```

Sorting Descending and Custom Sorting

Descending Sorting

- To sort in descending order, use `reverse=True` as an argument in `sort()`.

Example (Python)

```
thislist = ["orange","mango","kiwi","pineapple","banana"]
thislist.sort(reverse=True)
print(thislist)
# Output: ["pineapple","orange","mango","kiwi","banana"]
```

Sorting Descending and Custom Sorting contd.

Custom Sorting

- Customize sorting with your own function using `key`.

Example (Python)

```
def myfunc(n):
  return abs(n - 50)
thislist = [100, 50, 65, 82, 23]
thislist.sort(key=myfunc)
print(thislist)
# Output: [50, 65, 23, 82, 100]
```

Case-Insensitive and Reverse Sorting

Case-Insensitive Sorting

- Use `key=str.lower` for case-insensitive sorting.

Example (Python)

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key=str.lower)
print(thislist)
# Output: ["banana", "cherry", "Kiwi", "Orange"]
```

Case-Insensitive and Reverse Sorting contd.

Reverse Order

- Reverse the order of list items with the `reverse()` method.

Example (Python)

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
# Output: ["cherry", "Kiwi", "Orange", "banana"]
```

Copying a List

Copying a List

- Simply assigning one list to another (`list2 = list1`) creates a reference, not a copy.

Using the `copy()` Method

- To create a copy of a list, use the `copy()` method.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
# Output: ["apple", "banana", "cherry"]
```

Copying a List contd.

Using the `list()` Method

- Another way to create a copy is to use the `list()` method.

Example (Python)

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
# Output: ["apple", "banana", "cherry"]
```

Tuples in Python

What are Tuples?

- Tuples are used to store multiple items in a single variable.
- They are ordered and unchangeable.
- Written with round brackets.

Creating a Tuple

Example (Python)

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Tuples in Python contd.

Tuple Characteristics

- Ordered: Items have a defined order that won't change.
- Unchangeable: Items cannot be modified, added, or removed after creation.
- Allow Duplicates: Tuple can have items with the same value.

Tuple Length

Example (Python)

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Tuples in Python contd.

Creating a One-Item Tuple

Example (Python)

```
one_item_tuple = ("apple",)
print(type(one_item_tuple))
# Not a tuple
not_a_tuple = ("apple")
print(type(not_a_tuple))
```

Tuple Data Types and the tuple() Constructor

Tuple Data Types

Example (Python)

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

Tuple Data Type

- From Python's perspective, tuples are objects with the data type 'tuple'.

Example (Python)

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

Using the tuple() Constructor

Collections in Python

Python Collections

Python offers several collection data types:

- Lists: Ordered and changeable, allows duplicate members.
- Tuples: Ordered and unchangeable, allows duplicate members.
- Sets: Unordered, unchangeable, and unindexed, no duplicate members.
- Dictionaries: Ordered and changeable, no duplicate members.

Python Conditions and If Statements

Logical Conditions in Python

Python supports various logical conditions:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

Python Conditions and If Statements contd.

If Statements

An "if statement" is used to execute code based on a condition.

Example (Python)

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Python Conditions and If Statements contd.

Indentation in Python

Python uses indentation to define code scope, unlike other languages that use curly brackets.

Example (Python)

```
a = 33
b = 200
if b > a:
print("b is greater than a") # This will raise an error
```

elif and else Statements

elif Statement

The `elif` keyword is used to test another condition if the previous conditions were not true.

Example (Python)

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

elif and else Statements contd.

else Statement

The `else` keyword catches anything that wasn't caught by the preceding conditions.

Example (Python)

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Short Hand If Statements

One Line if Statement

If you have only one statement to execute, you can put it on the same line as the `if` statement.

Example (Python)

```
if a > b: print("a is greater than b")
```

Short Hand If Statements contd.

One Line if else Statement

For both `if` and `else`, you can put them all on the same line using a ternary operator.

Example (Python)

```
a = 2
b = 330
print("A") if a > b else print("B")
```

Logical Operators in Python

Using Logical Operators

Logical operators are used to combine conditional statements.

- `and`: Returns `True` if both conditions are `True`.
- `or`: Returns `True` if at least one condition is `True`.
- `not`: Reverses the result of the conditional statement.

Logical Operators in Python contd.

Examples of Logical Operators

Example (Python)

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
if a > b or a > c:
  print("At least one condition is True")
if not a > b:
  print("a is NOT greater than b")
```

Nested If Statements and the pass Statement

Nested If Statements

You can have if statements inside if statements; this is called nested if statements.

Example (Python)

```
x = 41
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

Nested If Statements and the pass Statement contd.

The pass Statement

If you have an empty if statement for some reason, use the pass statement to avoid errors.

Example (Python)

```
a = 33
b = 200
if b > a:
  pass
```

Python Loops

Python Loop Commands

Python provides two primary loop commands:

- `while` loops
- `for` loops

Python Loops contd.

The `while` Loop

The `while` loop executes a set of statements as long as a condition is true.

Example (Python)

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The `break` and `continue` Statements

The `break` Statement

The `break` statement allows you to exit the loop prematurely, even if the `while` condition is still true.

Example (Python)

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The break and continue Statements contd.

The continue Statement

The continue statement lets you skip the current iteration and move to the next one.

Example (Python)

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement

The else Statement with while

The else statement allows you to run a block of code once when the while condition is no longer true.

Example (Python)

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Python For Loops

Overview

Python's `for` loop is used for iterating over a sequence (like a list, tuple, dictionary, set, or string) and executing a set of statements for each item in the sequence.

Example: Looping Through a List

Loop through and print each fruit in a list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Looping Through a String

Example: Looping Through a String

Even strings are iterable. Loop through and print each letter in the word "banana":

```
for x in "banana":
  print(x)
```

The break Statement

With break, you can exit the loop prematurely. Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

The continue Statement

Example: The continue Statement

Use continue to skip the current iteration. In this case, do not print "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The continue Statement contd.

The range() Function

The range() function generates a sequence of numbers for looping. Loop through values from 0 to 5:

```
for x in range(6):
  print(x)
```

The range() Function (contd.)

Example: Specifying Start and Increment

You can specify the start and increment values. For example, loop through values from 2 to 6 (not including 6):

```
for x in range(2, 6):
  print(x)
```

The range() Function (contd.)

Example: Increment by 3

Increment the sequence by 3:

```
for x in range(2, 30, 3):
  print(x)
```

else in For Loop

The else Statement in a for Loop

The else statement in a for loop specifies a block of code to be executed when the loop is finished:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

else in For Loop contd.

else with break

The else block will NOT be executed if the loop is stopped by a break statement:

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

Nested Loops

Nested loops are loops inside loops. The inner loop is executed once for each iteration of the outer loop:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

Nested Loops contd.

The pass Statement

If a `for` loop has no content, you can use the `pass` statement to avoid errors:

```
for x in [0, 1, 2]:
  pass
```

Python Classes/Objects

Overview

Python is an object-oriented programming language. Almost everything in Python is an object, with its properties and methods. A class is like a blueprint for creating objects.

Create a Class

To create a class, use the `class` keyword:

```
class MyClass:
  x = 5
```

Create Object

Example: Create an Object

Now we can use the class named `MyClass` to create objects:

```
p1 = MyClass()
print(p1.x)
```

Create Object contd.

The `__init__()` Function

The `__init__()` function is called automatically every time a new object is created. It is used to initialize object properties:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

The __str__() Function

The __str__() Function

The __str__() function controls what is returned when the object is represented as a string:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def __str__(self):
    return f"{self.name}({self.age})"
p1 = Person("John", 36)
print(p1)
```

The __str__() Function contd.

Object Methods

Objects can contain methods (functions that belong to the object):

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello, my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

The self Parameter

The self Parameter

The self parameter is a reference to the current instance of the class and is used to access variables that belong to the class:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello, my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

Modify Object Properties

You can modify object properties like this:

```
p1.age = 40
```

Delete Object Properties

You can delete object properties using the `del` keyword:

```
del p1.age
```

Delete Objects

You can delete objects using the `del` keyword:

```
del p1
```

The pass Statement

The pass Statement

If a class definition is empty, you can use the pass statement to avoid getting an error:

```
class Person:
  pass
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. Parent class is the class being inherited from, also called base class. Child class inherits from another class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
#Create an obj using Person class, and exe the printname method:
x = Person("John", "Doe")
x.printname()
```

Python Inheritance

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

```
class Student(Person):
  pass
```

Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

Example

Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Python Inheritance

Add the `__init__()` Function

So far we have created a child class that inherits the properties and methods from its parent.
We want to add the `__init__()` function to the child class (instead of the pass keyword).

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

Note: `__init__()` function is called automatically every time the class is being used to create a new object.

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Python Inheritance

Add the `__init__()` Function

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Python Inheritance

Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Python Inheritance

Add Properties

Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2025
```

In the example below, the year 2025 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the `__init__()` function:

Python Inheritance

Add Properties

Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year
x = Student("Mike", "Olsen", 2025)
```

Python Inheritance

Add Methods

Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year
  def welcome(self):
    print("Welcome", self.firstname, self.lastname,
    "to the class of", self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Python Iterators

An iterator is an object that contains a countable number of values.
An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
All these objects have a iter() method which is used to get an iterator:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

Python Iterators

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Python Iterators

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)


mystr = "banana"
for x in mystr:
  print(x)
```

The for loop actually creates an iterator object and executes the next() method for each loop.

Python Iterators

Create an Iterator

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.
As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.
The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Python Iterators

Create an Iterator

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
  def __next__(self):
    x = self.a
    self.a += 1
    return x
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Python Iterators

StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.
To prevent the iteration from going on forever, we can use the StopIteration statement.
In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Python Iterators

StopIteration

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration
myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
  print(x)
```

Python Polymorphism

Understanding Polymorphism

- "Polymorphism" means "many forms" in programming.
- It refers to methods, functions, or operators with the same name that can be executed on different objects or classes.

Function Polymorphism

- An example of Python function polymorphism is the `len()` function.

Python Polymorphism Examples

```
# For strings, len() returns the number of characters.
x = "Hello World!"
print(len(x))
# For tuples, len() returns the number of items.
mytuple = ("apple", "banana", "cherry")
print(len(mytuple))
# For dictionaries, len() returns the number of key/value pairs.
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(len(thisdict))
```

Class Polymorphism

Polymorphism in Class Methods: Polymorphism is often used in class methods, allowing multiple classes to have the same method name.

```
# Different classes with the same method:
class Car:
  def move(self):
    print("Drive!")
class Boat:
  def move(self):
    print("Sail!")
class Plane:
  def move(self):
    print("Fly!")
car1 = Car()        # Create a Car class
boat1 = Boat()      # Create a Boat class
plane1 = Plane()    # Create a Plane class
for x in (car1, boat1, plane1):
  x.move()
```

Due to polymorphism, the same method can be executed for all three classes.

Inheritance and Class Polymorphism

- Classes with child classes can also use polymorphism.
- Child classes inherit methods from the parent class but can override them.

```
# Create a class called Vehicle and make Car, Boat, Plane child classes of Vehicle:
class Vehicle:
  def move(self):
    print("Move!")
class Car(Vehicle):
  pass
class Boat(Vehicle):
  def move(self):
    print("Sail!")
class Plane(Vehicle):
  def move(self):
    print("Fly!")
car1 = Car()        # Create a Car object
boat1 = Boat()      # Create a Boat object
plane1 = Plane()    # Create a Plane object
for x in (car1, boat1, plane1):
  x.move()
```

Child classes inherit properties and methods from the parent class, with

Python Scope

Understanding Scope in Python

- In Python, a variable's availability is determined by its scope.

Local Scope

- A variable created inside a function belongs to the local scope of that function and is only accessible within it.

Example: Local Scope

```
def myfunc():
  x = 300
  print(x)
myfunc()
```

The variable ‘x‘ is available inside the ‘myfunc()‘ function.

Python Scope (Contd.)

Function Inside Function

- A local variable can also be accessed by functions within the same function.

Example: Function Inside Function

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()
myfunc()
```

The local variable ‘x‘ can be accessed from ‘myinnerfunc()‘ within ‘myfunc()‘.

Global Scope

- Variables created in the main body of the code have global scope.
- Global variables are accessible from any scope, including both global and local.

Python Scope (Contd.)

Example: Global Scope

```
x = 300
def myfunc():
  print(x)
myfunc()
print(x)
```

The variable ‘x‘ is global and can be used by both ‘myfunc()‘ and in the global scope.

Naming Variables

- If you have the same variable name inside and outside a function, Python treats them as separate variables with different scopes.

Example: Naming Variables

```
x = 300
def myfunc():
  x = 200
  print(x)
myfunc()
print(x)
```

The function prints the local ‘x‘, and then the code prints the global ‘x‘

Python Scope (Contd.)

Global Keyword

- If you need to create a global variable within a local scope, you can use the global keyword.

Example: Using the Global Keyword

```
def myfunc():
  global x
  x = 300
myfunc()
print(x)
```

The ‘global‘ keyword makes the variable ‘x‘ global.

Example: Changing a Global Variable

```
x = 300
def myfunc():
  global x
  x = 200
myfunc()
print(x)
```

Use the ‘global‘ keyword to change the value of a global variable inside a function.

Python Modules

What is a Module?

- A module in Python is similar to a code library.
- It's a file containing a set of functions and variables you want to include in your application.

Creating a Module

- To create a module, save the code in a file with the .py file extension.

Example: Creating a Module

```
# Save this code in a file named mymodule.py
def greeting(name):
  print("Hello, " + name)
```

Python Modules (Contd.)

Using a Module

- To use the module, you can import it using the import statement.

Example: Using a Module

```
# Import the module named mymodule and call the greeting function
import mymodule
mymodule.greeting("Jonathan")
```

When using a function from a module, use the syntax: $module_name.function_name$.

Python Modules (Contd.)

Variables in a Module

- Modules can contain functions as well as variables of various types (arrays, dictionaries, objects, etc.).

Example: Variables in a Module

```
# Save this code in the file mymodule.py
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example: Accessing Variables in a Module

```
# Import the module named mymodule and access the person1 dictionary
import mymodule
a = mymodule.person1["age"]
print(a)
```

Python Modules (Contd.)

Naming and Re-naming Modules

- Module files can be named as you like, but they must have the .py file extension.
- You can create an alias for a module when importing it using the as keyword.

Example: Creating an Alias

```
# Create an alias for mymodule called mx
import mymodule as mx
a = mx.person1["age"]
print(a)
```

Python Modules (Contd.)

Built-in Modules

- Python provides several built-in modules that you can import whenever you need them.

Example: Using a Built-in Module

```
# Import and use the platform module
import platform
x = platform.system()
print(x)
```

Python Modules (Contd.)

Using the dir() Function

- The dir() function can be used to list all the function and variable names in a module.

Example: Listing Defined Names in a Module

```
# List all the defined names belonging to the platform module
import platform
x = dir(platform)
print(x)
```

The dir() function can be used on all modules, including those you create yourself.

Python Modules (Contd.)

Importing from a Module

- You can choose to import only specific elements from a module using the from keyword.

Example: Importing a Dictionary from a Module

```
# The module named mymodule contains a function and a dictionary
def greeting(name):
  print("Hello, " + name)
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example: Using the from Keyword

```
# Import only the person1 dictionary from the module
from mymodule import person1
print(person1["age"])
```

Python Datetime

Python Dates

- In Python, we can work with dates using the ‘datetime‘ module, which provides date objects.

Example: Importing the datetime Module

```
import datetime
x = datetime.datetime.now()
print(x)
```

The result will be the current date and time, like ”2023-11-05 15:36:12.538383”.

Python Datetime (Contd.)

Date Output

- A ‘datetime‘ object contains year, month, day, hour, minute, second, and microsecond.
- The module provides methods to extract information from date objects.

Example: Extracting Year and Weekday

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

You can extract the year and name of the weekday.

Python Datetime (Contd.)

Creating Date Objects

- You can create a date object using the ‘datetime()‘ constructor by specifying year, month, and day.

Example: Creating a Date Object

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

The ‘datetime()‘ constructor also supports optional parameters for time and timezone.

Python Datetime (Contd.)

The strftime() Method

- You can format date objects into readable strings using the ‘strftime()‘ method.
- It takes a format parameter to specify the output format.

Example: Formatting the Month Name

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

You can use various format codes to customize the output.

Python Math

Built-in Math Functions

- Python provides a set of built-in math functions to perform mathematical tasks on numbers.

Example: Finding Minimum and Maximum

```
x = min(5, 10, 25)
y = max(5, 10, 25)
print(x)
print(y)
```

You can use ‘min()‘ and ‘max()‘ to find the lowest and highest values in an iterable.

Python Math (Contd.)

Built-in Math Functions (Contd.)

- The ‘abs()‘ function returns the absolute (positive) value of a number.

Example: Finding Absolute Value

```
x = abs(-7.25)
print(x)
```

It returns the absolute value of -7.25.

Python Math (Contd.)

Built-in Math Functions (Contd.)

- The ‘pow(x, y)‘ function returns the value of ‘x‘ to the power of ‘y‘ (xy).

Example: Calculating Power

```
x = pow(4, 3)
print(x)
```

It calculates 4 to the power of 3.

Python Math (Contd.)

The Math Module

- Python also provides a built-in ‘math‘ module that extends the list of mathematical functions.
- To use it, you must import the ‘math‘ module.

Example: Using the math Module

```
import math
x = math.sqrt(64)
print(x)
```

It calculates the square root of 64 using ‘math.sqrt()‘.

Python Math (Contd.)

The Math Module (Contd.)

- The ‘math.ceil()‘ method rounds a number upwards to its nearest integer, while ‘math.floor()‘ rounds a number downwards.

Example: Rounding with ‘math.ceil()‘ and ‘math.floor()‘

```
import math
x = math.ceil(1.4)
y = math.floor(1.4)
print(x)  # returns 2
print(y)  # returns 1
```

‘math.ceil()‘ rounds up to 2, and ‘math.floor()‘ rounds down to 1.

Python Math (Contd.)

The Math Module (Contd.)

- The ‘math.pi‘ constant returns the value of PI (3.14...).

Example: Using ‘math.pi‘

```
import math
x = math.pi
print(x)
```

It returns the value of PI, approximately 3.14159265359.

Python JSON

JSON in Python

- JSON (JavaScript Object Notation) is a syntax for storing and exchanging data in text format.
- Python has a built-in package called json for working with JSON data.

Example: Importing the json Module

```
import json
```

You can import the ‘json‘ module to work with JSON data.

Python JSON (Contd.)

Parse JSON - Convert from JSON to Python

- To parse a JSON string into a Python object, use the ‘json.loads()‘ method.
- It returns a Python dictionary.

Example: Converting from JSON to Python

```
import json
# Some JSON data
x = ’{ "name":"John", "age":30, "city":"New York"}’
# Parse the JSON data
y = json.loads(x)
# Result is a Python dictionary
print(y["age"])
```

It converts JSON data into a Python dictionary.

Python JSON (Contd.)

Convert from Python to JSON

- You can convert Python objects into JSON strings using the ‘json.dumps()‘ method.
- Supported Python object types include dictionaries, lists, strings, numbers, and more.

Example: Converting from Python to JSON

```
import json
# A Python dictionary
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}
# Convert to JSON
y = json.dumps(x)
# Result is a JSON string
print(y)
```

It converts a Python dictionary into a JSON string.

Python JSON (Contd.)

Convert Python Objects to JSON

- You can convert various Python object types into JSON strings.

Example: Converting Different Python Objects

```
import json
print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas"))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

Python objects are converted into their JSON equivalents.

Python JSON (Contd.)

Formatting the Result

- The ‘json.dumps()‘ method can be customized to format the JSON result for readability.

Example: Using ‘json.dumps()‘ Parameters

```
import json
x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}
# Customize the formatting
result = json.dumps(x, indent=4, separators=(". ", " = "), sort_keys=True)
print(result)
```

You can customize the indentation, separators, and key sorting in the JSON result.

Python RegEx

What is RegEx?

- Regular Expression (RegEx) is a sequence of characters forming a search pattern.
- RegEx is used to check if a string contains a specified search pattern.

RegEx Module in Python

- Python provides a built-in package called ‘re‘ for working with Regular Expressions.

Example: Importing the re Module

```
import re
```

You can import the ‘re‘ module to use regular expressions.

Python RegEx (Contd.)

RegEx in Python

- After importing the ‘re‘ module, you can use regular expressions in Python.

Example: Searching for a Pattern

```
import re
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

The ‘re.search()‘ function is used to find a pattern.

In this example, we check if the string starts with "The" and ends with "Spain".

Python RegEx (Contd.)

RegEx Functions

- The ‘re‘ module offers several functions for working with regular expressions:
  - ‘findall‘: Returns a list containing all matches.
  - ‘search‘: Returns a Match object if there is a match anywhere in the string.
  - ‘split‘: Returns a list where the string has been split at each match.
  - ‘sub‘: Replaces one or many matches with a string.

Python RegEx (Contd.)

Metacharacters

- Metacharacters are characters with special meanings in RegEx.

Example: Common Metacharacters

```
[] \ . ^ $ * + ? {} | ()
```

These metacharacters have special meanings in RegEx patterns.

Python RegEx (Contd.)

Special Sequences

- Special sequences start with a backslash ‘followed by a character and have specific meanings.

Example: Common Special Sequences

```
\A \b \B \d \D \s \S \w \W \Z
```

These special sequences are used for matching specific patterns.

Python RegEx (Contd.)

Sets

- Sets are enclosed in square brackets ‘[]‘ and specify a set of characters to match.

Example: Using Sets in RegEx

```
[arn] [a-n] [^arn] [0123] [0-9] [0-5][0-9] [a-zA-Z] [+]
```

Sets allow you to specify character ranges and choices.

Python RegEx (Contd.)

The ‘findall()‘ Function

- The ‘findall()‘ function returns a list containing all matches in a string.

Example: Using ‘findall()‘

```
import re
txt = "The rain in Spain"
x = re.findall("ai", txt)
```

It returns all occurrences of the pattern as a list.

Python RegEx (Contd.)

The ‘search()‘ Function

- The ‘search()‘ function searches for a pattern and returns a Match object if there’s a match.

Example: Using ‘search()‘

```
import re
txt = "The rain in Spain"
x = re.search("\s", txt)
```

It returns a Match object with information about the first match.

Python RegEx (Contd.)

The ‘split()‘ Function

- The ‘split()‘ function splits a string at each match of the pattern and returns a list.

Example: Using ‘split()‘

```
import re
txt = "The rain in Spain"
x = re.split("\s", txt)
```

It splits the string at every white-space character.

Python RegEx (Contd.)

The ‘sub()‘ Function

- The ‘sub()‘ function replaces matches with a specified text.

Example: Using ‘sub()‘

```
import re
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
```

It replaces all white-space characters with the number 9.

Python RegEx (Contd.)

Match Object

- A Match Object contains information about the search and the result.

Example: Using Match Object

```
import re
txt = "The rain in Spain"
x = re.search("ai", txt)
```

The ‘x‘ object is a Match Object with properties and methods.

Python String Formatting

String Format() Method

- The ‘format()‘ method allows you to format selected parts of a string.
- You can use placeholders (curly brackets) to control the display of values within the text.

Example: Basic String Formatting

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

You can add values inside the curly brackets to specify how to convert the value.

Python String Formatting (Contd.)

Multiple Values

- You can format a string with multiple values by adding more values to the ‘format()‘ method.
- Correspondingly, add more placeholders in the text.

Example: Formatting with Multiple Values

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use placeholders for different values within the string.

Python String Formatting (Contd.)

Index Numbers

- You can use index numbers (inside curly brackets 0) to ensure values are placed in the correct placeholders.
- If you want to refer to the same value multiple times, use the index number.

Example: Using Index Numbers

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Index numbers help maintain the order of values in the string.

Python String Formatting (Contd.)

Named Indexes

- You can use named indexes by entering a name inside the curly brackets name.
- When passing parameter values, use the names with ‘txt.format(name=value)‘.

Example: Using Named Indexes

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname="Ford", model="Mustang"))
```

Named indexes provide a more descriptive way to format strings.

File Handling in Python

File Handling

- File handling is an essential aspect of web application development.
- Python provides various functions for file creation, reading, updating, and deletion.

The open() Function

- The key function for file operations in Python is `open()`.
- It accepts two parameters: *filename* and *mode*.

File Opening Modes

- "r" - Read (Default) - Opens a file for reading; raises an error if the file doesn't exist.
- "a" - Append - Opens a file for appending; creates the file if it doesn't exist.
- "w" - Write - Opens a file for writing; creates the file if it doesn't exist.
- "x" - Create - Creates the specified file; returns an error if it already exists.

File Handling in Python

File Handling File Mode Types

- "t" - Text (Default) - Text mode.
- "b" - Binary - Binary mode (e.g., for handling images).

Syntax

- To open a file for reading, simply specify the file name: `f = open("demofile.txt")`
- This is equivalent to: `f = open("demofile.txt", "rt")`
- Since "r" and "t" are the default values for read and text, you can omit them.

Opening and Reading Files in Python

Open a File on the Server

- To access a file, you can use the built-in `open()` function.
- The `open()` function returns a file object with a `read()` method.

Example: Opening and Reading a File

- Get your own Python Server

```
f = open("demofile.txt", "r")
print(f.read())
```

- If the file is in a different location, specify the file path.

Read Only Parts of the File

- By default, `read()` returns the whole text. You can specify the number of characters you want to read.

Example: Reading Specific Characters

- Return the first 5 characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Reading Lines and Closing Files

Read Lines

- You can retrieve one line at a time using the `readline()` method.

Example: Reading Lines

- Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

- You can read multiple lines by calling `readline()` iteratively.

Example: Looping Through the File

- Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Opening and Reading Files in Python

Close Files

- It's important to close files when you're done with them to ensure changes are saved.

Example: Closing the File

- Close the file when finished:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: Always close files to ensure changes are properly saved.

Writing to Files in Python

Write to an Existing File

- To write to an existing file, use the "a" (Append) or "w" (Write) mode with the `open()` function.

Example: Appending Content to a File

- Open the file "demofile2.txt" and append content:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()
```

- Open and read the file after appending.

Example: Overwriting Content in a File

- Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()
```

- Open and read the file after overwriting. Note that "w" will overwrite the entire file.

Creating New Files in Python

Create a New File

- To create a new file, use the `open()` method with specific parameters.

Example: Creating a New File

- Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

- Result: A new empty file is created if it doesn't exist.

Example: Creating a New File If It Doesn't Exist

- Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Deleting Files and Folders in Python

Delete a File

- To delete a file in Python, you need to import the OS module and use the `os.remove()` function.

Example: Deleting a File

```
import os
os.remove("demofile.txt")
```

This code will remove the file "demofile.txt" from the system.

Check if File Exists

- It's a good practice to check if the file exists before attempting to delete it to avoid errors.

Example: Checking and Deleting a File

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

This code first checks if the file exists, and if it does, it deletes it;

Let's learn by some examples

```
Some Bioinformatics Examples using basic Python functions
```

Counting Letters in DNA Strings

Given some string dna containing the letters A, C, G, or T, representing the bases that make up DNA, we ask the question: how many times does a certain base occur in the DNA string? For example, if DNA is ATGGCATTA and we ask how many times the base A occur in this string, the answer is 3.

Counting Letters in DNA Strings

A general Python implementation answering this problem can be done in many ways. Several possible solutions are presented below.

List Iteration

The most straightforward solution is to loop over the letters in the string, test if the current letter equals the desired one, and if so, increase a counter. Looping over the letters is obvious if the letters are stored in a list. This is easily done by converting a string to a list:

```
list('ATGC')
['A', 'T', 'G', 'C']
```

Our first solution becomes:

```
def count_v1(dna, base):
    dna = list(dna)  # convert string to list of letters
    i = 0            # counter
    for c in dna:
        if c == base:
            i += 1
    return i
```

Counting Letters in DNA Strings

Your complete code should look like this:

count_exp_1.py

```
1   def count_v1(dna, base):
2       dna = list(dna)          # convert string to list of letters
3
4       print(dna)               # Let's print our DNA sequence
5       print(base)              # Let's check what was our base of interest to count
6
7       i = 0                    # Set counter to zero to start counting
8       for c in dna:            # for loop
9           if c == base:        # if coundition, count IF c is equal to the base of interest
10              i = i + 1        # Increase the counter everytime the condition is satisfied
11      return i                 # return the counted number of an output of the function
12
13  dna = 'ATGCAAA'              # Test Sequence
14
15  n = count_v1(dna,'A')        # Keep counted number in a variable called n
16
17  print(n)                     # Print n
```

Counting Letters in DNA Strings

String Iteration

Python allows us to iterate directly over a string without converting it to a list. In fact, all built-in objects in Python that contain a set of elements in a particular sequence allow a for loop construction of the type for element in object. A slight improvement of our solution is therefore to iterate directly over the string:

```
def count_v2(dna, base):
    i = 0 # counter
    for c in dna:
        if c == base:
            i += 1
    return i
dna = 'ATGCGGACCTAT'
base = 'C'
n = count_v2(dna, base)
# printf-style formatting
print '%s appears %d times in %s' % (base, n, dna)
# or (new) format string syntax
print '{base} appears {n} times in {dna}'.format(
    base=base, n=n, dna=dna)
```

Program Flow

It is fundamental for correct programming to understand how to simulate a program by hand, statement by statement.

Printing variables and messages

Inserting print statements and examining the variables is the simplest approach to investigating what is going on:

```
def count_v2_demo(dna, base):
    print 'dna:', dna
    print 'base:', base
    i = 0 # counter
    for c in dna:
        print 'c:', c
        if c == base:
            print 'True if test'
            i += 1
    return i
n = count_v2_demo('ATGCGGACCTAT', 'C')
```

Index Iteration

Although it is natural in Python to iterate over the letters in a string (or more generally over elements in a sequence), programmers with experience from other languages (Fortran, C and Java are examples) are used to for loops with an integer counter running over all indices in a string or array:

```
def count_v3(dna, base):
    i = 0 # counter
    for j in range(len(dna)):
        if dna[j] == base:
            i += 1
    return i
```

Python indices always start at 0 so the legal indices for our string become 0, 1, ..., len(dna)-1, where len(dna) is the number of letters in the string dna. The range(x) function returns a list of integers 0, 1, ..., x-1, implying that range(len(dna)) generates all the legal indices for dna.

While Loops

The while loop equivalent to the last function reads

```
def count_v4(dna, base):
    i = 0 # counter
    j = 0 # string index
    while j < len(dna):
        if dna[j] == base:
            i += 1
        j += 1
    return i
```

Correct indentation is here crucial: a typical error is to fail indenting the j += 1 line correctly.

Summing a Boolean List

The idea now is to create a list m where m[i] is True if dna[i] equals the letter we search for (base). The number of True values in m is then the number of base letters in dna. We can use the sum function to find this number because doing arithmetics with boolean lists automatically interprets True as 1 and False as 0. That is, sum(m) returns the number of True elements in m. A possible function doing this is

```
def count_v5(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        if c == base:
            m.append(True)
        else:
            m.append(False)
    return sum(m)
```

Inline If Test

Shorter, more compact code is often a goal if the compactness enhances readability. The four-line if test in the previous function can be condensed to one line using the inline if construction: if condition value1 else value2.

```
def count_v6(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        m.append(True if c == base else False)
    return sum(m)
```

Using Boolean Values Directly

The inline if test is in fact redundant in the previous function because the value of the condition c == base can be used directly: it has the value True or False. This saves some typing and adds clarity, at least to Python programmers with some experience:

```
def count_v7(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        m.append(c == base)
    return sum(m)
```

List Comprehensions

Building a list with the aid of a for loop can often be condensed to a single line by using list comprehensions: [expr for e in sequence], where expr is some expression normally involving the iteration variable e. In our last example, we can introduce a list comprehension

```
def count_v8(dna, base):
    m = [c == base for c in dna]
    return sum(m)
```

Here it is tempting to get rid of the m variable and reduce the function body to a single line:

```
def count_v9(dna, base):
    return sum([c == base for c in dna])
```

Using a Sum Iterator

The DNA string is usually huge - 3 billion letters for the human species. Making a boolean array with True and False values therefore increases the memory usage by a factor of two in our sample functions `count_v5` to `count_v9`. Summing without actually storing an extra list is desirable. Fortunately, sum([x for x in s]) can be replaced by sum(x for x in s), where the latter sums the elements in s as x visits the elements of s one by one. Removing the brackets therefore avoids first making a list before applying sum to that list. This is a minor modification of the *countv9* function:

```
def count_v10(dna, base):
    return sum(c == base for c in dna)
```

Extracting Indices

Instead of making a boolean list with elements expressing whether a letter matches the given base or not, we may collect all the indices of the matches. This can be done by adding an if test to the list comprehension:

```
def count_v11(dna, base):
    return len([i for i in range(len(dna)) if dna[i] == base])
```

Try this in python shell:

```
>>> dna = 'AATGCTTA'
>>> base = 'A'
>>> indices = [i for i in range(len(dna)) if dna[i] == base]
>>> indices
[0, 1, 7]
>>> print dna[0], dna[1], dna[7]  # check
A A A
```

Observe that the element i in the list comprehension is only made for those i where dna[i] == base.

Generating Random DNA Strings

The simplest way of generating a long string is to repeat a character a large number of times:

```
N = 1000000
dna = 'A'*N
```

The resulting string is just 'AAA...A, of length N, which is fine for testing the efficiency of Python functions. Nevertheless, it is more exciting to work with a DNA string with letters from the whole alphabet A, C, G, and T. To make a DNA string with a random composition of the letters we can first make a list of random letters and then join all those letters to a string:

```
import random
alphabet = list('ATGC')
dna = [random.choice(alphabet) for i in range(N)]
dna = ''.join(dna)  # join the list elements to a string
```

Generating Random DNA Strings

The random.choice(x) function selects an element in the list x at random. Note that N is very often a large number. In Python version 2.x, range(N) generates a list of N integers. We can avoid the list by using xrange which generates an integer at a time and not the whole list. In Python version 3.x, the range function is actually the xrange function in version 2.x. Using xrange, combining the statements, and wrapping the construction of a random DNA string in a function, gives

```
import random
def generate_string(N, alphabet='ACGT'):
    return ''.join([random.choice(alphabet) for i in xrange(N)])
dna = generate_string(600000)
The call generate_string(10) may generate something like AATGGCAGAA.
```

Computing Frequencies

Your genetic code is essentially the same from you are born until you die, and the same in your blood and your brain. Which genes that are turned on and off make the difference between the cells. This regulation of genes is orchestrated by an immensely complex mechanism, which we have only started to understand. A central part of this mechanism consists of molecules called transcription factors that float around in the cell and attach to DNA, and in doing so turn nearby genes on or off. These molecules bind preferentially to specific DNA sequences, and this binding preference pattern can be represented by a table of frequencies of given symbols at each position of the pattern. More precisely, each row in the table corresponds to the bases A, C, G, and T, while column j reflects how many times the base appears in position j in the DNA sequence.

Generating Random DNA Strings

For example, if our set of DNA sequences are TAG, GGT, and GGG, the table becomes:

Base	0	1	2
A	0	1	0
C	0	0	0
G	2	2	2
T	1	0	1

From this table we can read that base A appears once in index 1 in the DNA strings, base C does not appear at all, base G appears twice in all positions, and base T appears once in the beginning and end of the strings. In the following we shall present different data structures to hold such a table and different ways of computing them. The table is known as a frequency matrix in bioinformatics and this is the term used here too.

Separate Frequency Lists

Since we know that there are only four rows in the frequency matrix, an obvious data structure would be four lists, each holding a row. A function computing these lists may look like

```
def freq_lists(dna_list):
    n = len(dna_list[0])
    A = [0]*n
    T = [0]*n
    G = [0]*n
    C = [0]*n
    for dna in dna_list:
        for index, base in enumerate(dna):
            if base == 'A':
                A[index] += 1
            elif base == 'C':
                C[index] += 1
            elif base == 'G':
                G[index] += 1
            elif base == 'T':
                T[index] += 1
    return A, C, G, T
```

Separate Frequency Lists

We need to initialize the lists with the right length and a zero for each element, since each list element is to be used as a counter. Creating a list of length n with object x in all positions is done by [x]*n. Finding the proper length is here carried out by inspecting the length of the first element in `dna_list`, the list of all DNA strings to be counted, assuming that all elements in this list have the same length.
In the for loop we apply the enumerate function, which is used to extract both the element value and the element index when iterating over a sequence. For example,

```
>>> for index, base in enumerate(['t', 'e', 's', 't']):
...   print index, base
...
0 t
1 e
2 s
3 t
```

Separate Frequency Lists

Here is a test,

```
dna_list = ['GGTAG', 'GGTAC', 'GGTGC']
A, C, G, T = freq_lists(dna_list)
print A
print C
print G
print T
```

Output:

```
[0, 0, 0, 2, 0]
[0, 0, 0, 0, 2]
[3, 3, 0, 1, 1]
[0, 0, 3, 0, 0]
```

Nested List

The frequency matrix can also be represented as a nested list M such that M[i][j] is the frequency of base i in position j in the set of DNA strings. Here i is an integer, where 0 corresponds to A, 1 to T, 2 to G, and 3 to C. The frequency is the number of times base i appears in position j in a set of DNA strings. Sometimes this number is divided by the number of DNA strings in the set so that the frequency is between 0 and 1. Note that all the DNA strings must have the same length.
The simplest way to make a nested list is to insert the A, C, G, and T lists into another list:

```
>>> frequency_matrix = [A, C, G, T]
>>> frequency_matrix[2][3]
2
>>> G[3]  # same element
2
```

Nested List

Alternatively, we can illustrate how to compute this type of nested list directly:

```
def freq_list_of_lists_v1(dna_list):
    # Create empty frequency_matrix[i][j] = 0
    # i=0,1,2,3 corresponds to A,T,G,C
    # j=0,...,length of dna_list[0]
    frequency_matrix = [[0 for v in dna_list[0]] for x in 'ACGT']
    for dna in dna_list:
      for index, base in enumerate(dna):
          if base == 'A':
              frequency_matrix[0][index] +=1
          elif base == 'C':
              frequency_matrix[1][index] +=1
          elif base == 'G':
              frequency_matrix[2][index] +=1
          elif base == 'T':
              frequency_matrix[3][index] +=1
    return frequency_matrix
```

Nested List

As in the case with individual lists we need to initialize all elements in the nested list to zero.
A call and printout,

```
dna_list = ['GGTAG', 'GGTAC', 'GGTGC']
frequency_matrix = freq_list_of_lists_v1(dna_list)
print frequency_matrix
```

Results:

```
[[0, 0, 0, 2, 0], [0, 0, 0, 0, 2], [3, 3, 0, 1, 1], [0, 0, 3, 0, 0]]
```

Dictionary for More Convenient Indexing

The series of if tests in the Python function `freq_list_of_lists_v1` are somewhat cumbersome, especially if we want to extend the code to other bioinformatics problems where the alphabet is larger. What we want is a mapping from base, which is a character, to the corresponding index 0, 1, 2, or 3. A Python dictionary may represent such mappings:

```
>>> base2index = {'A': 0, 'C': 1, 'G': 2, 'T': 3}
>>> base2index['G']
2
```

With the base2index dictionary we do not need the series of if tests and the alphabet 'ATGC' could be much larger without affecting the length of the code:

```
def freq_list_of_lists_v2(dna_list):
    frequency_matrix = [[0 for v in dna_list[0]] for x in 'ACGT']
    base2index = {'A': 0, 'C': 1, 'G': 2, 'T': 3}
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base2index[base]][index] += 1
    return frequency_matrix
```

NumPy Introduction

What is NumPy?

- NumPy is a Python library used for working with arrays.
- It provides functions for linear algebra, Fourier transforms, and matrices.
- Created by Travis Oliphant in 2005, it's an open source project.
- NumPy stands for "Numerical Python."

Why Use NumPy?

- Python has lists, but they are slow for certain tasks.
- NumPy's array object (ndarray) is up to 50x faster than Python lists.
- ndarray provides functions for easy array manipulation.
- Widely used in data science where speed and resources matter.

NumPy Introduction (Contd.)

Why is NumPy Faster Than Lists?

- NumPy arrays are stored in memory efficiently, unlike lists.
- This efficient memory layout is called "locality of reference."
- NumPy is optimized to work with modern CPU architectures.

Which Language is NumPy Written In?

- NumPy is a Python library.
- Most performance-critical parts are written in C or C++ for speed.

NumPy Introduction (Contd.)

Where is the NumPy Codebase?

- The source code for NumPy is hosted on GitHub.
- GitHub allows collaborative work on the same codebase.
- Repository URL: `https://github.com/numpy/numpy`

NumPy Getting Started

Installation of NumPy

- If you have Python and PIP already installed, NumPy installation is straightforward.
- Use the command: `pip install numpy`
- Consider using Python distributions like Anaconda or Spyder if installation fails.

Importing NumPy

- Once installed, import NumPy using the `import` keyword.
- Example:

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
```

NumPy Getting Started (Contd.)

Alias for NumPy

- NumPy is often imported with the alias `np`.
- Create an alias with the `as` keyword while importing.
- Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

Checking NumPy Version

- You can check the NumPy version using the `__version__` attribute.
- Example:

```
import numpy as np
print(np.__version__)
```

NumPy Creating Arrays

Creating a NumPy Array

- NumPy arrays are created using the `array()` function.
- You can pass a list, tuple, or array-like object to create an ndarray.
- Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

Dimensions in Arrays

- Arrays can have different dimensions, from 0-D (Scalars) to higher dimensions.
- Each dimension represents a level of array depth.

NumPy Array Dimensions

0-D Arrays (Scalars)

- 0-D arrays represent individual elements in an array.
- Example:

```
import numpy as np
arr = np.array(42)
```

1-D Arrays

- 1-D arrays have 0-D arrays as elements.
- They are the most common and basic arrays.
- Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

NumPy Array Dimensions (Contd.)

2-D Arrays

- 2-D arrays have 1-D arrays as elements.
- Often used to represent matrices or 2nd order tensors.
- Example:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
```

3-D Arrays

- 3-D arrays have 2-D arrays (matrices) as elements.
- Used to represent 3rd order tensors.
- Example:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]])
```

NumPy Array Dimensions (Contd.)

Checking Number of Dimensions

- NumPy arrays provide the `ndim` attribute to check the number of dimensions.
- Example:

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Higher Dimensional Arrays

- Arrays can have any number of dimensions.
- Use the `ndmin` argument to specify the number of dimensions.
- Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
```

NumPy Array Indexing

Access Array Elements

- Array indexing is the same as accessing an array element.
- Indexes in NumPy arrays start at 0.

Example - Get the first element:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Example - Get the second element:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[1])
```

NumPy Array Indexing (Contd.)

Example - Access third and fourth elements and add them:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[2] + arr[3])
```

Access 2-D Arrays

- Access elements using comma-separated integers for dimensions and indices.
- Think of 2-D arrays like tables with rows and columns.

Example - Access 1st row, 2nd column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

NumPy Array Indexing (Contd.)

Example - Access 2nd row, 5th column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10])
print('5th element on 2nd row: ', arr[1, 4])
```

Access 3-D Arrays

- Access elements using comma-separated integers for dimensions and indices.

Example - Access the third element:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]])
print(arr[0, 1, 2])
```

NumPy Array Indexing (Contd.)

Example Explained

- `arr[0, 1, 2]` prints the value 6.
- This is because the first number selects the first dimension, then the second selects the second dimension, and the third selects the third dimension.

Negative Indexing

- Use negative indexing to access elements from the end.

Example - Print the last element from the 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10])
print('Last element from 2nd dim: ', arr[1, -1])
```

NumPy Array Slicing

Slicing Arrays

- Slicing in Python means taking elements from one index to another.
- Format: `[start:end]` or `[start:end:step]`.
- Default values: `0` for `start`, length for `end`, and `1` for `step`.

Example - Slice elements from index 1 to 5:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Note: The result includes the start index but excludes the end index.

NumPy Array Slicing (Contd.)

Example - Slice elements from index 4 to the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

Example - Slice elements from beginning to index 4 (not included):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

NumPy Array Slicing (Contd.)

Negative Slicing

- Use the minus operator to refer to an index from the end.

Example - Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

NumPy Array Slicing (Contd.)

STEP

- Use the step value to determine the slicing step.

Example - Return every other element from index 1 to 5:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

Example - Return every other element from the entire array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[::2])
```

NumPy Array Slicing (Contd.)

Slicing 2-D Arrays

Example - From the second element, slice elements from index 1 to 4 (not included):

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

Note: The second element has index 1.

Example - From both elements, return index 2:

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10])
print(arr[0:2, 2])
```

NumPy Array Slicing (Contd.)

Example - From both elements, slice index 1 to 4 (not included), this will return a 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

NumPy Data Types

Data Types in Python

- Strings - used to represent text data in quotes (e.g., "ABCD").
- Integer - used to represent integer numbers (e.g., -1, -2, -3).
- Float - used to represent real numbers (e.g., 1.2, 42.42).
- Boolean - used to represent True or False.
- Complex - used to represent complex numbers (e.g., 1.0 + 2.0j, 1.5 + 2.5j).

Data Types in NumPy

- NumPy introduces additional data types represented by characters (e.g., i for integers, u for unsigned integers).

NumPy Data Types

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float

NumPy Data Types (Contd.)

Checking the Data Type of an Array

Example - Get the data type of an array object:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

Example - Get the data type of an array containing strings:

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

NumPy Data Types (Contd.)

Creating Arrays With a Defined Data Type

Example - Create an array with data type string:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

Example - Create an array with data type 4 bytes integer:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='i4')
print(arr)
print(arr.dtype)
```

NumPy Data Types (Contd.)

What if a Value Can Not Be Converted?
Example - A non-integer string like 'a' can't be converted to integer (will raise an error):

```
import numpy as np
arr = np.array(['a', '2', '3'], dtype='i')
```

Converting Data Type on Existing Arrays
Example - Change data type from float to integer using 'i' as a parameter:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

NumPy Data Types (Contd.)

Example - Change data type from float to integer using int as a parameter:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)
print(newarr)
print(newarr.dtype)
```

Example - Change data type from integer to boolean:

```
import numpy as np
arr = np.array([1, 0, 3])
newarr = arr.astype(bool)
print(newarr)
print(newarr.dtype)
```

NumPy Array Copy vs View

The Difference Between Copy and View

- The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
- The copy owns the data, and changes made to it won't affect the original array.
- The view does not own the data, and changes made to it will affect the original array.

NumPy Array Copy (COPY)

Example - Make a copy, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

The copy SHOULD NOT be affected by changes to the original array.

NumPy Array View (VIEW)

Example - Make a view, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

The view SHOULD be affected by changes made to the original array.

NumPy Array View (Make Changes in the VIEW)

Example - Make a view, change the view, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31
print(arr)
print(x)
```

The original array SHOULD be affected by changes made to the view.

NumPy Array Ownership Check

Check if Array Owns its Data

As mentioned above, copies own the data, and views do not own the data. How can we check this?

Every NumPy array has the attribute `base` that returns `None` if the array owns the data. Otherwise, the `base` attribute refers to the original object.

Example - Print the value of the `base` attribute to check if an array owns its data or not:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
y = arr.view()
print(x.base)
print(y.base)
```

The copy returns `None`, and the view returns the original array.

NumPy Array Shape

Shape of an Array

- The shape of an array is the number of elements in each dimension.

Get the Shape of an Array

NumPy arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements.
Example - Print the shape of a 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8])
print(arr.shape)
```

The example above returns (2, 4), indicating that the array has 2 dimensions with the first dimension having 2 elements and the second having 4.

What Does the Shape Tuple Represent?

Integers at every index tell about the number of elements the corresponding dimension has. In the example above, at index-4, we have the value 4, so we can say that the 5th (4 + 1 th) dimension has 4 elements.

Example - Create an array with 5 dimensions using `ndmin` and a vector with values 1,2,3,4:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('shape of array :', arr.shape)
```

NumPy Array Reshaping

Reshaping arrays

- Reshaping means changing the shape of an array.
- The shape of an array is the number of elements in each dimension.
- By reshaping, we can add or remove dimensions or change the number of elements in each dimension.

Reshape From 1-D to 2-D

Example - Convert a 1-D array with 12 elements into a 2-D array with 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Reshape From 1-D to 3-D

Example - Convert a 1-D array with 12 elements into a 3-D array with 2 arrays, each containing 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Can We Reshape Into Any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes. Example - Try converting a 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

Returns Copy or View?

Example - Check if the returned array is a copy or a view:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print(arr.reshape(2, 4).base)
```

The example above returns the original array, so it is a view.

Unknown Dimension

You are allowed to have one "unknown" dimension. Example - Convert a 1D array with 8 elements to a 3D array with 2x2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

Flattening the Arrays

Flattening an array means converting a multidimensional array into a 1D array. Example - Convert the array into a 1D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6])
newarr = arr.reshape(-1)
print(newarr)
```

Note

There are many functions for changing the shapes of arrays in numpy, including flatten, ravel, and also for rearranging the elements like rot90, flip, fliplr, flipud, etc. These fall under the Intermediate to Advanced section of numpy.

NumPy Array Iterating

Iterating Arrays

- Iterating means going through elements one by one.
- For 1-D arrays, it will go through each element one by one.

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

Iterating 2-D Arrays

In a 2-D array, it will go through all the rows.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6])
for x in arr:
  print(x)
```

Iterating Arrays Using nditer()

The function nditer() is a helping function that can be used from very basic to very advanced iterations. Example - Iterate through a 3-D array:

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]])
for x in np.nditer(arr):
  print(x)
```

Iterating Array With Different Data Types

We can use
op$_d$ *typesargumentandpassittheexpecteddatatypetochangethedatatypeofeleme*

```
import numpy as np

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

for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S'])
  print(x)
\end{verbnobox}
```

Iterating With Different Step Size

We can use filtering and followed by iteration. Example - Iterate through every scalar element of the 2D array, skipping 1 element:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for x in np.nditer(arr[:, ::2]):
  print(x)
```

Enumerated Iteration Using ndenumerate()

Enumeration means mentioning the sequence number of elements one by one. Example - Enumerate on 1D array elements:

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Enumerated Iteration Using ndenumerate()

Example - Enumerate on 2D array elements:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

NumPy Joining Arrays

Joining NumPy Arrays

- Joining means putting contents of two or more arrays in a single array.
- We join arrays by axes in NumPy.

Using concatenate()

Join two arrays using `concatenate()`:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.concatenate((arr1, arr2))
print(arr)
```

Using concatenate() with 2-D Arrays

Join two 2-D arrays along rows (axis=1):

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
```

Joining Arrays Using Stack Functions

Stacking is similar to concatenation, but it's done along a new axis.
Example - Stack two 1-D arrays along the second axis:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.stack((arr1, arr2), axis=1)
print(arr)
```

Stacking Along Rows

NumPy provides a helper function: `hstack()` to stack along rows.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2)
print(arr)
```

Stacking Along Columns

NumPy provides a helper function: `vstack()` to stack along columns.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack((arr1, arr2)
print(arr)
```

Stacking Along Height (depth)

NumPy provides a helper function: `dstack()` to stack along height (depth).

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.dstack((arr1, arr2))
print(arr)
```

NumPy Splitting Arrays

Splitting NumPy Arrays

- Splitting is the reverse operation of Joining.
- Joining merges multiple arrays into one, and Splitting breaks one array into multiple.
- We use `array_split()` for splitting arrays, specifying the array and the number of splits.

Using array_split()

Split the array in 3 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
```

Using array_split() - Contd

Split the array in 4 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 4)
print(newarr)
```

Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays. Split the 2-D array into three 2-D arrays.

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
newarr = np.array_split(arr, 3)
print(newarr)
```

Splitting 2-D Arrays - Contd

Split the 2-D array into three 2-D arrays (each containing 3 elements).

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16,
newarr = np.array_split(arr, 3)
print(newarr)
```

Splitting 2-D Arrays Along Rows

You can specify which axis to split along. Split the 2-D array into three 2-D arrays along rows.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16,
newarr = np.array_split(arr, 3, axis=1)
print(newarr)
```

Alternate Solution - `hsplit()`

Use `hsplit()` to split the 2-D array into three 2-D arrays along rows.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16,
newarr = np.hsplit(arr, 3)
print(newarr)
```

NumPy Searching Arrays

Searching Arrays

- You can search an array for a certain value and return the indexes that match using the `where()` method.

Using `where()`

Find the indexes where the value is 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

Using where() - Contd

Find the indexes where the values are even:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

Using where() - Contd

Find the indexes where the values are odd:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 1)
print(x)
```

Search Sorted

There is a method called `searchsorted()` which performs a binary search in the array. Find the indexes where the value 7 should be inserted:

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

Search Sorted - Contd

Search From the Right Side Find the indexes where the value 7 should be inserted, starting from the right:

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side='right')
print(x)
```

Multiple Values

To search for more than one value, use an array with the specified values. Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np
arr = np.array([1, 3, 5, 7])
x = np.searchsorted(arr, [2, 4, 6])
print(x)
```

NumPy Sorting Arrays

Sorting Arrays

- Sorting means putting elements in an ordered sequence.
- The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

Using sort()

Sort the array:

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr)
```

Using sort() - Contd

You can also sort arrays of strings, or any other data type:

```
import numpy as np
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

Using sort() - Contd

Sort a boolean array:

```
import numpy as np
arr = np.array([True, False, True])
print(np.sort(arr))
```

Sorting a 2-D Array

If you use the `sort()` method on a 2-D array, both arrays will be sorted:

```
import numpy as np
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

NumPy Filtering Arrays

Filtering Arrays

- Getting some elements out of an existing array and creating a new array out of them is called filtering.
- In NumPy, you filter an array using a boolean index list.

Filtering with Boolean Index

Example - Creating an array from selected elements:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr)
```

Creating the Filter Array

Example - Creating a filter array based on conditions:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element is higher than 42, set the value to True, otherwise False:
  if element > 42:
    filter_arr.append(True)
  else:
    filter_arr.append(False)
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Creating Filter Directly From Array

Example - Creating a filter array directly from the array:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Creating Filter Directly From Array - Contd

Example - Creating a filter array for even elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
filter_arr = arr % 2 == 0
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Random Numbers in NumPy

What is a Random Number?

- Random number does NOT mean a different number every time. Random means something that cannot be predicted logically.
- Computers work on programs, and programs are a definitive set of instructions. So there must be some algorithm to generate a random number.
- Random numbers generated through a generation algorithm are called pseudo random.
- Truly random numbers require an outside source of random data.

Generate Random Number

Example - Generate a random integer from 0 to 100:

```
from numpy import random
x = random.randint(100)
print(x)
```

Generate Random Float

Example - Generate a random float from 0 to 1:

```
from numpy import random
x = random.rand()
print(x)
```

Generate Random Array (Integers)

Example - Generate a 1-D array containing 5 random integers from 0 to 100:

```
from numpy import random
x = random.randint(100, size=(5))
print(x)
```

Generate Random Array (Integers) - Contd

Example - Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

```
from numpy import random
x = random.randint(100, size=(3, 5))
print(x)
```

Generate Random Array (Floats)

Example - Generate a 1-D array containing 5 random floats:

```
from numpy import random
x = random.rand(5)
print(x)
```

Generate Random Array (Floats) - Contd

Example - Generate a 2-D array with 3 rows, each row containing 5 random numbers:

```
from numpy import random
x = random.rand(3, 5)
print(x)
```

Generate Random Number From Array

Example - Return one of the values in an array:

```
from numpy import random
x = random.choice([3, 5, 7, 9])
print(x)
```

Generate Random Number From Array - Contd

Example - Generate a 2-D array that consists of the values in the array parameter (3, 5, 7, and 9):

```
from numpy import random
x = random.choice([3, 5, 7, 9], size=(3, 5))
print(x)
```

Random Data Distribution

What is Data Distribution?

- Data Distribution is a list of all possible values and how often each value occurs.
- Important when working with statistics and data science.

Random Distribution

Random Distribution

- A random distribution is a set of random numbers that follow a certain probability density function.
- Probability Density Function: Describes continuous probability, i.e., probability of all values in an array.

Random Distribution - Contd

Generating Random Numbers with Defined Probabilities

- We can generate random numbers based on defined probabilities using the ‘choice()‘ method of the random module.
- The ‘choice()‘ method allows us to specify the probability for each value.
- Probability is set by a number between 0 and 1, where 0 means the value will never occur, and 1 means the value will always occur.

Random Distribution - Example

Example - Generating a 1-D array with defined probabilities:

```
from numpy import random
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(100))
print(x)
```

Random Distribution - Example Contd

Example - Generating a 2-D array with defined probabilities:

```
from numpy import random
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(3, 5))
print(x)
```

Random Permutations of Elements

Random Permutations of Elements

- A permutation refers to an arrangement of elements. For example, [3, 2, 1] is a permutation of [1, 2, 3] and vice-versa.
- The NumPy Random module provides two methods for this: ‘shuffle()‘ and ‘permutation()‘.

Shuffling Arrays

Shuffling Arrays

- Shuffle means changing the arrangement of elements in-place, i.e., in the array itself.

Example:

```
from numpy import random
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
random.shuffle(arr)
print(arr)
```

Generating Permutation of Arrays

Generating Permutation of Arrays

- Use the ‘permutation()‘ method to generate a random permutation of elements in an array.

Example:

```
from numpy import random
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(random.permutation(arr))
```

Visualize Distributions With Seaborn

Visualize Distributions With Seaborn

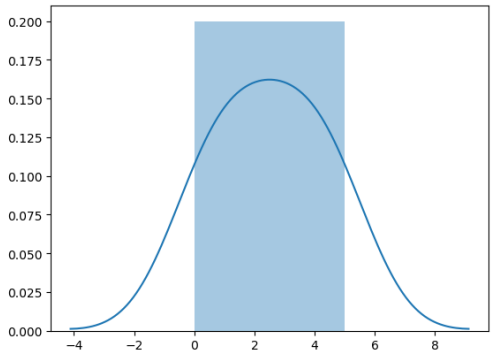


Figure: Enter Caption

- Seaborn is a library that uses Matplotlib underneath to plot graphs. It will be used to visualize random distributions.
- To get started, install Seaborn using the following command:

Install Seaborn

```
pip install seaborn
```

Distplots

Distplots

- Distplot stands for distribution plot, and it takes as input an array and plots a curve corresponding to the distribution of points in the array.

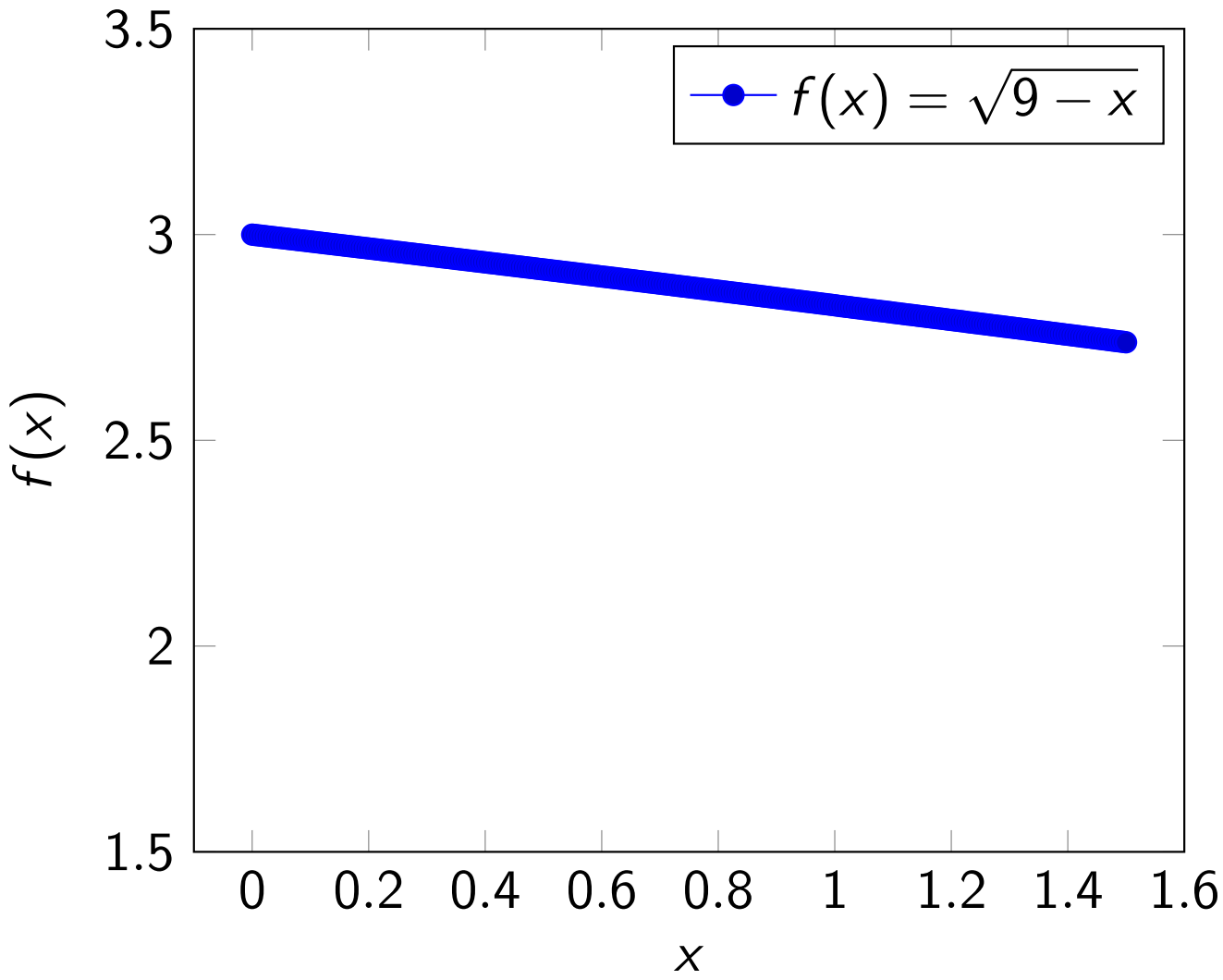
Import Matplotlib

```
import matplotlib.pyplot as plt
```

Import Seaborn

```
import seaborn as sns
```

Frame Title



(a) Caption of a figure.

Plotting a Distplot

Plotting a Distplot

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot([0, 1, 2, 3, 4, 5])
plt.show()
```

Plotting a Distplot Without the Histogram

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot([0, 1, 2, 3, 4, 5], hist=False)
plt.show()
```

Normal (Gaussian) Distribution

Normal (Gaussian) Distribution

- The Normal Distribution is one of the most important distributions, also known as the Gaussian Distribution.
- It fits the probability distribution of many events, e.g., IQ Scores, Heartbeat, etc.
- Use the `random.normal()` method to get a Normal Data Distribution with parameters:
  - `loc` - Mean (where the peak of the bell exists).
  - `scale` - Standard Deviation (how flat the graph distribution should be).
  - `size` - The shape of the returned array.

Example: Generate Normal Distribution

```
from numpy import random
x = random.normal(size=(2, 3))
print(x)
```

Example: Generate Normal Distribution with Mean and Standard Deviation

Example: Generate Normal Distribution with Mean and Standard Deviation

```
from numpy import random
x = random.normal(loc=1, scale=2, size=(2, 3))
print(x)
```

Example: Visualization of Normal Distribution

Example: Visualization of Normal Distribution

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.normal(size=1000), hist=False)
plt.show()
```

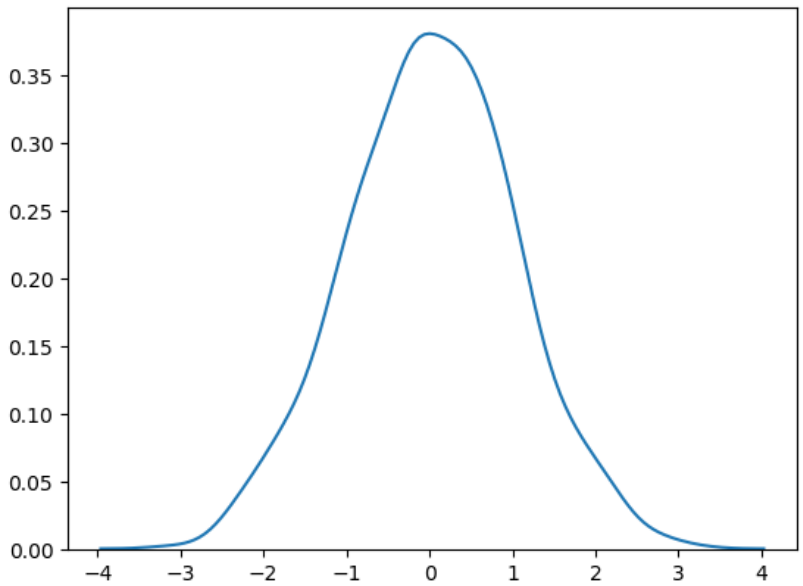


Figure: Enter Caption

Exponential Distribution

Exponential Distribution

- Exponential distribution is used for describing time until the next event, e.g., failure/success, etc.
- It has two parameters:
  - `scale` - Inverse of rate (default to 1.0).
  - `size` - The shape of the returned array.

Example: Draw Sample from Exponential Distribution

```
from numpy import random
x = random.exponential(scale=2, size=(2, 3))
print(x)
```

Example: Visualization of Exponential Distribution

Example: Visualization of Exponential Distribution

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.exponential(size=1000), hist=False)
plt.show()
```

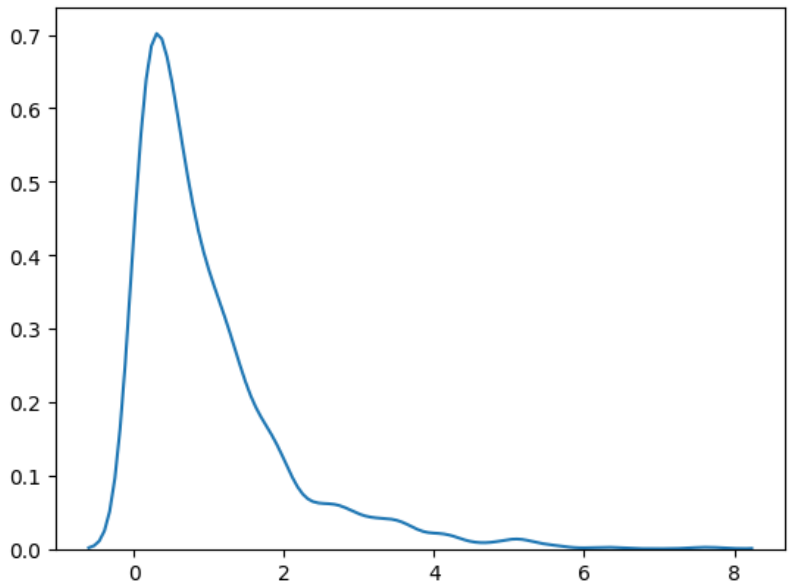


Figure: Enter Caption

NumPy ufuncs

What are ufuncs?

- ufuncs stand for "Universal Functions" and are NumPy functions that operate on the ndarray object.

Why use ufuncs?

Why use ufuncs?

- ufuncs are used to implement vectorization in NumPy, which is way faster than iterating over elements.
- They provide broadcasting and additional methods like reduce, accumulate, etc. that are very helpful for computation.
- ufuncs also take additional arguments, like:
  - `where` - boolean array or condition defining where the operations should take place.
  - `dtype` - defining the return type of elements.
  - `out` - output array where the return value should be copied.

What is Vectorization?

What is Vectorization?

- Converting iterative statements into a vector-based operation is called vectorization.
- It is faster as modern CPUs are optimized for such operations.

Example: Adding Elements of Two Lists

Example: Adding Elements of Two Lists

```
list 1: [1, 2, 3, 4]
list 2: [4, 5, 6, 7]
```

One way of doing it is to iterate over both of the lists and then sum each element.

```
Without ufunc, we can use Python's built-in zip() method:
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []
for i, j in zip(x, y):
  z.append(i + j)
print(z)
```

Example: Adding Elements of Two Lists (continued)

Example: Adding Elements of Two Lists (continued) NumPy has a ufunc for this, called `add(x, y)` that will produce the same result.

```
With ufunc, we can use the \texttt{add()} function:
import numpy as np
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = np.add(x, y)
print(z)
```

How To Create Your Own ufunc

How To Create Your Own ufunc

- To create your own ufunc, you have to define a function, like you do with normal functions in Python, then you add it to your NumPy ufunc library with the `frompyfunc()` method.
- The `frompyfunc()` method takes the following arguments:
  - `function` - the name of the function.
  - `inputs` - the number of input arguments (arrays).
  - `outputs` - the number of output arrays.

Example: Creating Your Own ufunc for Addition

Example: Creating Your Own ufunc for Addition

```
Create your own ufunc for addition:
import numpy as np
def myadd(x, y):
  return x + y
myadd = np.frompyfunc(myadd, 2, 1)
print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
```

Check if a Function is a ufunc

Check if a Function is a ufunc

- Check the type of a function to see if it is a ufunc or not.
- A ufunc should return `<class 'numpy.ufunc'>`.

Example: Checking if a Function is a ufunc

Example: Checking if a Function is a ufunc

```
Check if a function is a ufunc:
import numpy as np
print(type(np.add))
```

If it is not a ufunc, it will return another type, like this built-in NumPy function for joining two or more arrays.

Example: Checking if a Function is a ufunc (continued)

Example: Checking if a Function is a ufunc (continued)

```
Check the type of another function: concatenate():
import numpy as np
print(type(np.concatenate))
```

If the function is not recognized at all, it will produce an error.

Example: Checking if a Non-Existent Function is a ufunc

Example: Checking if a Non-Existent Function is a ufunc

Check the type of something that does not exist. This will produce an error:

```
import numpy as np
print(type(np.blahblah))
```

Example: Using an if Statement to Check if a Function is a ufunc

Example: Using an if Statement to Check if a Function is a ufunc

Use an if statement to check if the function is a ufunc or not:

```
import numpy as np
if type(np.add) == np.ufunc:
  print('add is ufunc')
else:
  print('add is not ufunc')
```

How To Create Your Own ufunc

How To Create Your Own ufunc

- To create your own ufunc, you have to define a function, like you do with normal functions in Python, then you add it to your NumPy ufunc library with the `frompyfunc()` method.
- The `frompyfunc()` method takes the following arguments:
  - `function` - the name of the function.
  - `inputs` - the number of input arguments (arrays).
  - `outputs` - the number of output arrays.

Example: Creating Your Own ufunc for Addition

Example: Creating Your Own ufunc for Addition

```
Create your own ufunc for addition:
import numpy as np
def myadd(x, y):
  return x + y
myadd = np.frompyfunc(myadd, 2, 1)
print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
```

Check if a Function is a ufunc

Check if a Function is a ufunc

- Check the type of a function to see if it is a ufunc or not.
- A ufunc should return `<class 'numpy.ufunc'>`.

Example: Checking if a Function is a ufunc

Example: Checking if a Function is a ufunc

```
Check if a function is a ufunc:
import numpy as np
print(type(np.add))
```

If it is not a ufunc, it will return another type, like this built-in NumPy function for joining two or more arrays.

Example: Checking if a Function is a ufunc (continued)

```
Check the type of another function: concatenate():
import numpy as np
print(type(np.concatenate))
```

If the function is not recognized at all, it will produce an error.

Example: Checking if a Non-Existent Function is a ufunc

Check the type of something that does not exist. This will produce an error:

```
import numpy as np
print(type(np.blahblah))
```

Example: Using an if Statement to Check if a Function is a ufunc

Example: Using an if Statement to Check if a Function is a ufunc

Use an if statement to check if the function is a ufunc or not:

```
import numpy as np
if type(np.add) == np.ufunc:
  print('add is ufunc')
else:
  print('add is not ufunc')
```

Simple Arithmetic

Simple Arithmetic

- You could use arithmetic operators + - * / directly between NumPy arrays, but this section discusses an extension of the same where we have functions that can take any array-like objects, e.g., lists, tuples, etc., and perform arithmetic conditionally.
- Arithmetic Conditionally: means that we can define conditions where the arithmetic operation should happen.

Addition

Addition

- The `add()` function sums the content of two arrays and returns the results in a new array.

Example: Addition

Example: Addition

Add the values in arr1 to the values in arr2:

```
import numpy as np
arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.add(arr1, arr2)
print(newarr)
```

Subtraction

Subtraction

- The `subtract()` function subtracts the values from one array with the values from another array and returns the results in a new array.

Example: Subtraction

Example: Subtraction

Subtract the values in arr2 from the values in arr1:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.subtract(arr1, arr2)
print(newarr)
```

Multiplication

Multiplication

- The `multiply()` function multiplies the values from one array with the values from another array and returns the results in a new array.

Example: Multiplication

Multiply the values in arr1 with the values in arr2:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.multiply(arr1, arr2)
print(newarr)
```

Division

Division

- The `divide()` function divides the values from one array with the values from another array and returns the results in a new array.

Example: Division

Example: Division

Divide the values in arr1 with the values in arr2:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 10, 8, 2, 33])
newarr = np.divide(arr1, arr2)
print(newarr)
```

Power

Power

- The `power()` function raises the values from the first array to the power of the values of the second array and returns the results in a new array.

Example: Power

Example: Power

Raise the values in arr1 to the power of values in arr2:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 6, 8, 2, 33])
newarr = np.power(arr1, arr2)
print(newarr)
```

Remainder

Remainder

- Both the `mod()` and the `remainder()` functions return the remainder of the values in the first array corresponding to the values in the second array and return the results in a new array.

Example: Remainder

Example: Remainder

```
Return the remainders:
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])
newarr = np.mod(arr1, arr2)
print(newarr)
```

Quotient and Mod

Quotient and Mod

- The `divmod()` function returns both the quotient and the mod. The return value is two arrays, the first array contains the quotient and the second array contains the mod.

Example: Quotient and Mod

Example: Quotient and Mod

```
Return the quotient and mod:
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])
newarr = np.divmod(arr1, arr2)
print(newarr)
```

The first array represents the quotients, and the second array represents the remainders of the same divisions.

Matplotlib Tutorial

What is Matplotlib?

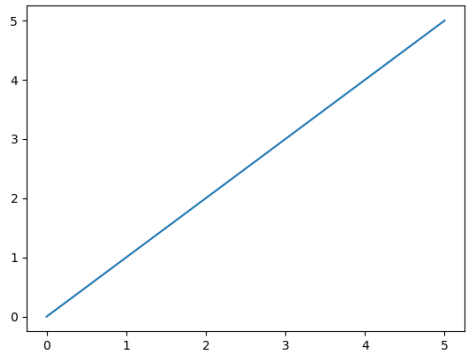


Figure: Enter Caption

- Matplotlib is a low-level graph plotting library in Python that serves as a visualization utility.
- Created by John D. Hunter.
- Open source and freely available.
- Mostly written in Python, with some segments in C, Objective-C, and JavaScript for platform compatibility.

Matplotlib Codebase

Matplotlib Codebase

- The source code for Matplotlib is available on GitHub at the following repository: `https://github.com/matplotlib/matplotlib`

Matplotlib Getting Started

Installation of Matplotlib

- If you have Python and PIP already installed on your system, you can easily install Matplotlib using the following command:

```
C:\Users\Your Name>pip install matplotlib
```

- If this command fails, consider using a Python distribution that already includes Matplotlib, such as Anaconda or Spyder.

Matplotlib Getting Started (Contd.)

Import Matplotlib

- After installing Matplotlib, import it into your applications by adding the import module statement:
  `import matplotlib`
- Now Matplotlib is imported and ready to use in your Python programs.

Matplotlib Getting Started (Contd.)

Checking Matplotlib Version

- You can check the Matplotlib version by accessing the version string stored under the `__version__` attribute.
  Example:

```
import matplotlib
print(matplotlib.__version__)
```

Matplotlib Pyplot

Pyplot

- Most of the Matplotlib utilities are available under the `pyplot` submodule, and they are usually imported using the `plt` alias.
- Import the `pyplot` package as `plt` for convenience.
  Example:
  ```
  import matplotlib.pyplot as plt
  ```

Matplotlib Pyplot (Contd.)

Example:

- You can use `plt` to create visualizations easily. For instance, to draw a line from position (0,0) to position (6,250):

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
plt.plot(xpoints, ypoints)
plt.show()
```

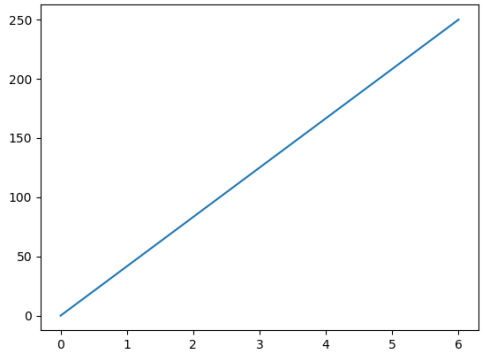


Figure: Enter Caption

Matplotlib Markers

Markers

- Use the keyword argument `marker` to emphasize each point with a specified marker.

  Example:

```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, marker = 'o')
plt.show()
```

Matplotlib Markers (Contd.)

Marker Reference

- `'o'`: Circle
- `'*'`: Star
- `'.'`: Point
- `','`: Pixel
- `'x'`: X
- `'X'`: X (filled)
- `'+'`: Plus
- `'P'`: Plus (filled)
- `'s'`: Square
- `'D'`: Diamond
- `'d'`: Diamond (thin)
- `'p'`: Pentagon

Matplotlib Markers (Contd.)

Marker Reference (Contd.)

- `'H'`: Hexagon
- `'h'`: Hexagon
- `'v'`: Triangle Down
- `'^'`: Triangle Up
- `'<'`: Triangle Left
- `'>'`: Triangle Right
- `'1'`: Tri Down
- `'2'`: Tri Up
- `'3'`: Tri Left
- `'4'`: Tri Right
- `'|'`: Vline
- `'_'`: Hline

Matplotlib Markers (Contd.)

Format Strings (fmt)

- You can use the shortcut string notation parameter to specify the marker.
- This parameter is also called `fmt`, and is written with this syntax: `marker|line|color`.
  Example:
  `plt.plot(ypoints, 'o:r')`

Matplotlib Markers (Contd.)

Marker Size

- You can use the keyword argument `markersize` or the shorter version, `ms` to set the size of the markers.
  Example:
  ```
  plt.plot(ypoints, marker = 'o', ms = 20)
  ```

Matplotlib Markers (Contd.)

Marker Color

- You can use the keyword argument `markeredgecolor` or the shorter `mec` to set the color of the edge of the markers.
  Example:
  `plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r')`

Matplotlib Markers (Contd.)

Marker Color (Contd.)

- You can use the keyword argument `markerfacecolor` or the shorter `mfc` to set the color inside the edge of the markers.
  Example:
  ```
  plt.plot(ypoints, marker = 'o', ms = 20, mfc = 'r')
  ```

Matplotlib Markers (Contd.)

Marker Color (Contd.)

- Use both the `mec` and `mfc` arguments to color the entire marker.
  Example:
  ```
  plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r', mfc = 'r')
  ```

Matplotlib Markers (Contd.)

Marker Color (Contd.)

- You can also use Hexadecimal color values.
  Example:

```
plt.plot(ypoints, marker = 'o', ms = 20, mec =
'#4CAF50', mfc = '#4CAF50')
```

Matplotlib Markers (Contd.)

Marker Color (Contd.)

- Or any of the 140 supported color names.
  Example:
  ```
  plt.plot(ypoints, marker = 'o', ms = 20, mec =
  'hotpink', mfc = 'hotpink')
  ```

Matplotlib Subplot

Display Multiple Plots

- With the `subplot()` function, you can draw multiple plots in one figure.

  Example:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x, y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x, y)
plt.show()
```

Matplotlib Subplot (Contd.)

The `subplot()` Function

- The `subplot()` function takes three arguments that describe the layout of the figure.
- The layout is organized in rows and columns, which are represented by the first and second argument.
- The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
plt.subplot(1, 2, 2)
```

Matplotlib Subplot (Contd.)

The `subplot()` Function (Contd.)

- So, if we want a figure with 2 rows and 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this.
  Example:

```
plt.subplot(2, 1, 1)
plt.subplot(2, 1, 2)
```

Matplotlib Subplot (Contd.)

Super Title

- You can add a title to the entire figure with the `suptitle()` function.
  Example:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x, y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x, y)
plt.suptitle("MY SHOP")
plt.show()
```

Matplotlib Histograms

Histogram

- A histogram is a graph showing frequency distributions.
- It displays the number of observations within each given interval.
- Example: Say you ask for the height of 250 people, you might end up with a histogram like this:
  - 2 people from 140 to 145cm
  - 5 people from 145 to 150cm
  - 15 people from 151 to 156cm
  - ...
  - 21 people from 185 to 190cm
  - 4 people from 190 to 195cm

Matplotlib Histograms (Contd.)

Create Histogram

- In Matplotlib, we use the `hist()` function to create histograms.
- The `hist()` function takes an array of numbers to create a histogram.
- For simplicity, we use NumPy to randomly generate an array with 250 values, where the values concentrate around 170, and the standard deviation is 10.

  Example:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.normal(170, 10, 250)
plt.hist(x)
plt.show()
```

Matplotlib Histograms (Contd.)

Create Histogram

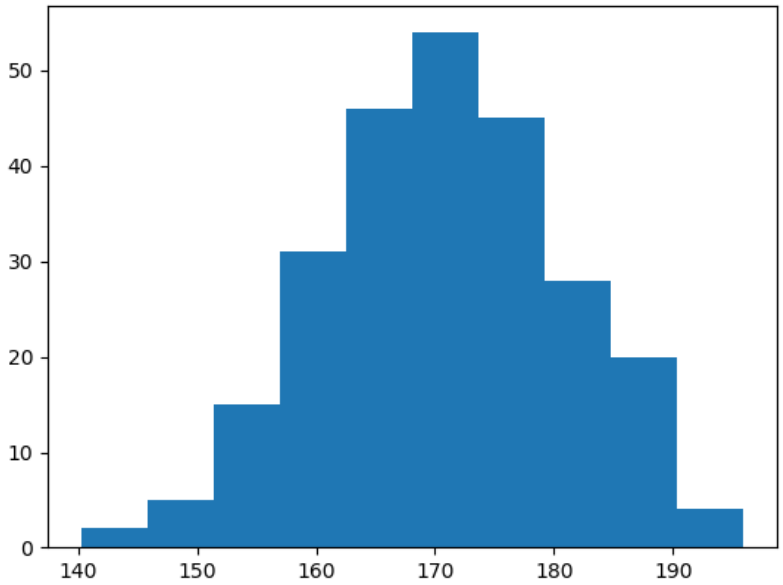


Figure: Enter Caption

Machine Learning

What is Machine Learning?

- Machine Learning involves making computers learn from data and statistics.
- It's a step toward artificial intelligence (AI).
- Machine Learning programs analyze data and learn to predict outcomes.

Where to Start with Machine Learning

- In this tutorial, we'll revisit mathematics and delve into statistics.
- We'll learn how to calculate important statistics based on data sets.
- We'll explore Python modules to obtain the answers we need.
- Additionally, we'll develop functions to predict outcomes based on what we've learned.

Data Set in Machine Learning

- In the context of a computer, a data set can be any collection of data, ranging from an array to a complete database.
- Examples:
  - Array: [99,86,87,88,111,86,103,87,94,78,77,85,86]
  - Database:

Carname	Color	Age	Speed	AutoPass
BMW	red	5	99	Y
Volvo	black	7	86	Y
VW	gray	8	87	N
VW	white	7	88	Y
Ford	white	2	111	Y
VW	white	17	86	Y
Tesla	red	2	103	Y
BMW	black	9	87	Y
Volvo	gray	4	94	N
Ford	white	11	78	N
Toyota	gray	12	77	N
VW	white	9	85	N
Toyota	blue	6	86	Y

- Machine Learning aims to analyze data and make predictions based on that analysis.

Data Types in Machine Learning

- To analyze data effectively, we need to understand the data types we're dealing with.
- Main data types:
  - Numerical
  - Categorical
  - Ordinal
- Numerical data can be divided into:
  - Discrete Data (e.g., count of cars passing by)
  - Continuous Data (e.g., price or size of an item)
- Categorical data can't be measured directly (e.g., color or yes/no values).
- Ordinal data can be ranked (e.g., school grades where A is better than B).
- Understanding your data type helps determine the right analysis techniques.
- We'll explore statistics and data analysis in the following chapters.

Mean, Median, and Mode

Exploring Numerical Data

In Machine Learning, three key values often interest us when analyzing a group of numbers:

- **Mean** - The average value.
- **Median** - The middle point value.
- **Mode** - The most common value.

Mean in Machine Learning

Calculating the Mean

To calculate the mean, sum all values and divide by the number of values. For example:

$$\frac{99 + 86 + 87 + 88 + 111 + 86 + 103 + 87 + 94 + 78 + 77 + 85 + 86}{13} \approx 89.77$$

You can use the NumPy module to calculate the mean.

Median in Machine Learning

Finding the Median

The median is the middle value when all numbers are sorted.

- First, sort the numbers.
- If there's an even count of numbers, take the average of the two middle values.

You can use NumPy to find the median.

Mode in Machine Learning

Identifying the Mode

The mode is the value that appears most frequently in the dataset.

For example, in the dataset: 99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86, the mode is 86.

You can use the SciPy module to find the mode.

Chapter Summary

Key Takeaways

Mean, Median, and Mode are fundamental techniques in Machine Learning. Understanding these concepts is crucial for data analysis and predictive modeling.

Standard Deviation in Machine Learning

Understanding Standard Deviation

Standard deviation is a metric that quantifies the spread of data values.

- Low standard deviation: Data points are close to the mean.
- High standard deviation: Data points are spread over a wider range.

Standard Deviation Example

Example 1

Given the speed of 7 cars:

$$\text{speed} = [86, 87, 88, 86, 87, 85, 86]$$

The standard deviation is approximately 0.9.

This means most values are within 0.9 units of the mean, which is 86.4.

Standard Deviation Example (Contd.)

Example 2

Consider a different set of speeds:

$$\text{speed} = [32, 111, 138, 28, 59, 77, 97]$$

The standard deviation is approximately 37.85.
This indicates a wider spread of values, with most falling within 37.85 units of the mean (77.4).

Using NumPy for Standard Deviation

Calculating Standard Deviation with NumPy

The NumPy module provides a method to compute the standard deviation.

- Example 1:

```
import numpy

speed = [86, 87, 88, 86, 87, 85, 86]

x = numpy.std(speed)

print(x)
```

- Example 2:

```
import numpy

speed = [32, 111, 138, 28, 59, 77, 97]

x = numpy.std(speed)

print(x)
```

Summary of Standard Deviation

Key Takeaways

Standard deviation is a vital metric in Machine Learning. It helps assess the spread of data values. Understanding this concept is crucial for data analysis and modeling.

Questions?