Python to Advance

Disclaimer and Copyright Statement

- This document is a training document. Therefore, it should not be considered as a system specification.
- Due to continuous advances in methods, design, and manufacturing, the contents of this document are subject to revision without notice.
- Ericsson assumes no responsibility for any errors or damages resulting from reference to this document.
- This document is not intended to replace the technical documentation that came with the system. Always refer to that technical documentation during operation and maintenance.

Course objectives

After completing this course, students will be able to:

- Using data structures more flexibly
- Writing more complexed functions
- Practicing Object-Oriented Programming
- Working with python libraries

Data Structures

Python to Advance

Data Model

- **Object**
- *Objects* are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.
- Every object has an identity, a type and a value.

- **Identity**
  - An object's *identity* never changes once it has been created; (object's address in memory)
  - The ***is*** operator compares the identity of two objects;
  - The ***id()*** function returns an integer representing its identity.
- **Type**
  - Determines the operations that the object supports and also defines the possible values for objects of that type.
  - The ***type()*** function returns an object's type.
- **Value**
  - Objects whose value can change are said to be *mutable*;
  - Objects whose value is unchangeable once they are created are called *immutable*.

Casting – Specify a Variable Type

- Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.
  - int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
  - float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
  - str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

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

-
```
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
```

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

Python Collections

- There are four collection data types in the Python programming language:
  - List is a collection which is ordered and changeable. Allows duplicate members.
  - Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
  - Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
  - Dictionary is a collection which is ordered** and changeable. No duplicate members.

Nested List

- A nested list is a list that contains other lists.
- Representing multi-dimensional data structures: You can use nested lists to represent data that has multiple dimensions. For example, you can use nested lists to represent a matrix or a table.
- Storing collections of collections: Nested lists can be used to store collections of collections. For example, you can use nested lists to represent a tree data structure.

```
matrix = [
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
]

print(matrix[1][2])
```

Nested Dictionary

- A nested dictionary is a dictionary that contains other dictionaries.
- Storing hierarchical data: Nested dictionaries can be used to store hierarchical data. For example, you can use nested dictionaries to represent a configuration file or a nested JSON object.
- Organizing related data: Nested dictionaries can be used to organize related data. For example, you can use nested dictionaries to represent a database table with multiple levels of relationships.

```
users = {
   'user1': { 'name': 'John', 'age': 30, 'city': 'New York' },
   'user2': { 'name': 'Jane', 'age': 25, 'city': 'San Francisco' }
}
user = {
            "name": "John",
            "age": 25,
            "address": {
                        "street": "123 Main St",
                        "city": "New York",
                        "state": "NY"
}}
```


Nested List and Dictionary

- Combination of Nested Lists and Dictionaries:
- Representing complex data structures: You can combine nested lists and nested dictionaries to represent complex data structures. For example, you can use nested lists to represent a list of objects, and nested dictionaries to represent the properties of each object.
- Storing hierarchical and related data: Combining nested lists and nested dictionaries can be used to store both hierarchical and related data. For example, you can use nested lists to represent a tree structure, and nested dictionaries to represent the properties of each node in the tree.

```
books = [
   {'title': 'Book 1', 'author': 'Author 1', 'price': 10.99},
   {'title': 'Book 2', 'author': 'Author 2', 'price': 12.99},
   {'title': 'Book 3', 'author': 'Author 3', 'price': 9.99}
]

for book in books:
   print(book['title'], book['author'], book['price'])
```

Sequences

- These are finite ordered sets indexed by non-negative numbers.
- Sequences support slicing: (a[i:j] selects all items with index k such that i <= k < j.)
  - a slice is a sequence of the same type.
- Sequences are distinguished according to their mutability:
  - Immutable sequences : Strings, Tuples, Bytes
    - An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)
  - Mutable sequences : Lists, Byte Arrays
    - Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and del (delete) statements.

Copy a List

- Shallow Copy:
  - Use the built-in list method copy()
  - Use the built-in method list()
  - Use the "slice" operator
- Deep Copy:
  - Use deepcopy() from copy

- list1 = ["apple", "banana", "cherry"]
  mylist = list1.copy()
- list1 = ["apple", "banana", "cherry"]
  mylist = list(list1)
- list1 = ["apple", "banana", "cherry"]
  mylist = list1[:]

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
shallow_copy = original_list.copy()
deep_copy = copy.deepcopy(original_list)
```

Sort a List

- Use the built-in list method sort()
  - It modifies the list in-place and return None
- Use the built-in function sorted()
  - It returns a new sorted list
  - Accepts any iterable
- Both list.sort() and sorted() have a key parameter to specify a function to be called on each list element prior to making comparisons

```
list1 = [100, 50, 65, 82, 23]
list1.sort()
list1.sort(reverse=True)

def myfunc(n):
  return abs(n - 50)
list1.sort(key = myfunc)

mylist = sorted(list1)
mylist = sorted(list1, key=myfunc)
```


Comprehension

- Comprehensions provide a concise way to create a list, a set or a dictionary.
- Usage:
  - To make new list/set/dictionary where each element is the result of some operations applied to each member of another sequence or iterable
  - To create a subsequence of those elements that satisfy a certain condition.
- Benefits:
  - Readability: Comprehensions make your code more readable. They allow you to create complex data structures in a single line of code.
  - Conciseness: Comprehensions are more concise than traditional for loops.
  - Efficiency: Comprehensions are generally more efficient than using traditional for loops. They are optimized for performance.
  - Flexibility: It can be used with various data structures, such as lists, sets, and dictionaries.
  - Code Golf: It help you write code in fewer lines, which is useful for code golfing and code challenges.

List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an iterable.
- The *iterable* can be any iterable object, like a list, tuple, set etc.
- newlist = [*expression* for *item* in *iterable* if *condition* == True]

- List comprehension is considerably faster than processing a list using the for loop.

- fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = []

  for x in fruits:
  if "a" in x:
  newlist.append(x)

- fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

  newlist = [x for x in fruits if "a" in x]

List Comprehension

- Examples of using list comprehension

```
vec = [-4, -2, 0, 2, 4]

# create a new list with the values doubled
[x*2 for x in vec]
[-8, -4, 0, 4, 8]


# filter the list to exclude negative numbers
[x for x in vec if x >= 0]
[0, 2, 4]


# apply a function to all the elements
[abs(x) for x in vec]
[4, 2, 0, 2, 4]
```

```
# call a method on each element
freshfruit = ['  banana', '  loganberry ', 'passion
fruit  ']

[weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']

# create a list of 2-tuples like (number, square)
[(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

# flatten a list using a listcomp with two 'for'
vec = [[1,2,3], [4,5,6], [7,8,9]]
[num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Set Comprehension

- Set comprehension is a concise way to create sets.
- For example, you can use a set comprehension to create a set of unique numbers:

```
numbers = {x for x in [1, 2, 3, 2, 1, 4, 5, 4, 3, 1]}
print(numbers)
```


Dictionary Comprehension

- Dictionary comprehension is a concise and readable way to create dictionaries.
- For example, you can use a dictionary comprehension to create a dictionary with keys and values:

```
squares = {x: x**2 for x in range(10)}
print(squares)
```

Nested Comprehension

- You can use nested comprehensions to create complex data structures:

```
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
]

transposed = [[row[i] for row in matrix]
for i in range(4)]

print(transposed)
```

```
data = {
   'group1': [1, 2, 3],
   'group2': [4, 5, 6],
   'group3': [7, 8, 9]
}

result = {group: [x**2 for x in values] for group, values in
data.items()}
print(result)
```


Functions

Python to Advance

Documentation Strings

- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*.
- There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code;
- it's good practice to include docstrings in code that you write, so make a habit of it.

```
def my_function():
    """Do nothing, but document it.


    No, really, it doesn't do anything.

    """

    pass


print(my_function.__doc__)
# Do nothing, but document it.

#No, really, it doesn't do anything.
```


Local or Global?

- When you create a variable inside a function, it is local, which means that it only exists inside the function.
- When you create a variable outside of any function is known as global variables.
- To create a global variable inside a function, you can use the **global** keyword; also, use the **global** keyword if you want to change a global variable inside a function.

```
x = "awesome"

def myfunc():
    # global x
    x = "fantastic"
    print("Python is " + x)

myfunc()
print("Python is " + x)
```

Parameters or Arguments?

- **Parameters** are defined by the names that appear in a function definition
- **Arguments** are the values passed to a function when calling it
- Parameters define what types of arguments a function can accept.

```
def my_function(name: str):
    print("Welcome " + name)

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```


Default Argument Values

- The most useful form is to specify a default value for one or more arguments.
- This creates a function that can be called with fewer arguments than it is defined to allow.

```
def ask_ok(prompt, retries=4, reminder="Please try again!"):
    while True:
        reply = input(prompt)
        if reply in {"y", "ye", "yes"}:
            return True
        if reply in {"n", "no", "nop", "nope"}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError("invalid user response")
        print(reminder)
```

Keyword Arguments

- Functions can also be called using keyword arguments of the form kwarg=value.
- This way the order of the arguments does not matter.
- Keyword arguments must follow positional arguments.
- All the keyword arguments passed must match one of the arguments accepted by the function, and their order is not important.

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```


Special parameters

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

- where / and * are optional.
- Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
- Use keyword-only when names have meaning, and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
- For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a tuple of arguments and can access the items accordingly.
- Before the variable number of arguments, zero or more normal arguments may occur.
- Write the function call with the * operator to unpack the arguments out of a list or tuple.

```
def my_function(*kids):
    print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
args = ("Emil", "Tobias", "Linus")
my_function(*args)
```


Arbitrary Keyword Arguments, **kwargs

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
- This way the function will receive a dictionary of arguments and can access the items accordingly.
- Dictionaries can deliver keyword arguments with the **-operator.

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

```
d = {"fname": "Tobias", "lname": "Refsnes"}
my_function(**d)
```

Lambda Expressions

- Small anonymous functions can be created with the **lambda** keyword.
- **lambda** ***arguments*** **:** ***expression***
- They are syntactically restricted to a single expression.
- Semantically, they are just syntactic sugar for a normal function definition.
- Like nested function definitions, lambda functions can reference variables from the containing scope

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```


Recursion

- Recursion is a common mathematical and programming concept. It means that a function calls itself.
- This has the benefit of meaning that you can loop through data to reach a result.
- Correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power.

```
def collatz(number, step=1):
    if number == 1:
        print('final result will be 1 no mater what positive integer you enter!!!')
        return 1
    elif number % 2 == 0:
        number = number // 2
    else:
        number = number * 3 + 1
    print(f'Step: {step:>3}, number = {number:>5}')
    step += 1
    return collatz(number, step)
```

Higher-Order Functions

- In Python, higher-order functions are functions that can take other functions as arguments or return functions as outputs. They allow you to manipulate functions as if they were regular values.

1. **Functions as Arguments:** Higher-order functions can take other functions as arguments. This allows you to pass functions as parameters to a function and use them within the function.
2. **Functions as Return Values:** Higher-order functions can return functions as outputs. This allows you to create and return functions dynamically.
3. **Function Composition:** Higher-order functions can be used to compose functions together. This means you can create new functions by combining existing functions.
4. **Functional Programming:** Higher-order functions are a key concept in functional programming. They enable you to write code in a more declarative and compositional style.

```
def apply_function(func, value):
    return func(value)


def square(x):
    return x**2


result = apply_function(square, 5)

print(result)
```


filter()

- The filter() function in Python is a built-in function that takes a function and an iterable (e.g., list, tuple, or string) as arguments.
- Return an iterator yielding those items of iterable for which function(item) is true; If function is None, return the items that are true.
- ***filter***(function, iterable)
  - function: This is the function that will be applied to each item of the iterable.
  - iterable: This is the sequence of items to which the function will be applied.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even_numbers))
```

```
words = ["apple", "banana", "cherry", "date",
"elderberry"]

long_words = filter(lambda word: len(word) > 5,
words)

print(list(long_words))
```

map()

- The map() function in Python is a built-in function that applies a given function to each item of an iterable (e.g., list, tuple, or string) and returns an iterator.
- ***map***(function, iterable, /, * iterables)
  - function: This is the function that you want to apply to each item of the iterable.
  - iterable: This is the sequence of items to which the function will be applied.

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = map(lambda x: x**2, numbers)

print(list(squared_numbers))
```

```
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

sum_of_numbers = map(lambda x, y: x + y, numbers1,
numbers2)

print(list(sum_of_numbers))
```


reduce()

- The reduce() function in Python is a built-in function from the functools module. It applies a given function to the items of an iterable (e.g., list, tuple, or string) and returns a single value.
- ***from functools import reduce***
- ***reduce***(function, iterable[, initializer], /)
  - function: This is the function that you want to apply to each item of the iterable.
  - iterable: This is the sequence of items to which the function will be applied.
  - initializer (optional): This is the initial value for the accumulator. It is used as the starting value for the first application of the function.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

sum_of_numbers = reduce(lambda x, y: x + y,
numbers)

print(sum_of_numbers)
```

```
numbers = [1, 5, 3, 8, 2, 6]

max_number = reduce(lambda x, y: x if x > y else y,
numbers)

print(max_number)
```

Object-Oriented Programming

Python to Advance

Python Class/Instance Objects

- Python is an object-oriented programming language. Almost everything in Python is an object, with its attributes and methods.
- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

```
# create a class object

class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

```
# create an instance object

x = MyClass()
```

```
# operate objects

print(x.i)
print(x.f())
```

__init__() and self

- **__init__()**
  - When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly created instance.
  - Use __init__() function to assign values to object attributes, or other operations that are necessary when the object is being created.
- **self**
  - The **self** parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
  - It must be the first parameter of any function object that is a class attribute defines a method for instances of that class.

```
# create a class

class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, targetX, targetY):
        return ((self.x - targetX)**2 + (self.y - targetY)**2)**0.5
```

```
# create instance objects

p1 = Point(3, 4)
p2 = Point(5, 6)
```

```
# operation of objects

print(p1.distance(0, 0))
print(p1.distance(p2.x, p2.y))
```


Class and Instance variables

- Instance variables
  - Instance variables are for data unique to each instance.
  - They are defined within the __init__() method or any other method of the class.
- Class variables
  - Class variables are for attributes and methods shared by all instances of the class.
  - They are defined within the class definition, outside of any methods

```
class MyClass:
    def __init__(self, instance_var):
        self.instance_var = instance_var

obj1 = MyClass("Instance variable 1")
obj2 = MyClass("Instance variable 2")

print(obj1.instance_var)
print(obj2.instance_var)
```

```
class MyClass:
    class_var = "This is a class variable"

    def __init__(self, instance_var):
        self.instance_var = instance_var

obj1 = MyClass("Instance variable 1")
obj2 = MyClass("Instance variable 2")

print(MyClass.class_var)
print(obj1.class_var)
print(obj2.class_var)
```

Object Operations

- Modify object attributes
- Delete object attributes
- Delete 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()
```

```
# modify object attributes

p1.age = 40
```

```
# delete object (attributes)

del p1.age
del p1
```


Inheritance - Concepts

- Inheritance is a mechanism that allows one class to inherit the properties and behaviors of another class. It allows you to create a new class that is a modified version of an existing class.
- Parent Class: The class from which the new class inherits is called the parent class.
- Child Class: The new class that inherits from the parent class is called the child class.
- Inheritance Syntax: To create a child class that inherits from a parent class, you use the following syntax:

```
# syntax for inheritance

class ChildClass(ParentClass):
  # class definition
```

```
class Animal:
    def __init__(self, name):
        self.name = name

    def eat(self):
        print(f"{self.name} is eating")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking")

dog = Dog("Rex", "Golden Retriever")
dog.eat()
dog.bark()
```

Inheritance – Checking Tools

- isinstance()
  - isinstance() function is used to check if an object is an instance of a class or a subclass.
  - isinstance(object, classinfo)
- issubclass()
  - issubclass() function is used to check if one class is a subclass of another class.
  - issubclass(class, classinfo)
    - class: This is the class that you want to check.
    - classinfo: This is the class or a tuple of classes to check against.

```
class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
        print(f"Calculating the area of {self.name}")

class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def area(self):
        print(f"The area of {self.name} is {self.length *
self.width}")

rectangle = Rectangle("Square", 5, 5)
rectangle.area()

print(isinstance(rectangle, Rectangle))
print(isinstance(rectangle, Shape))
print(isinstance(rectangle, int))

print(issubclass(Rectangle, Shape))
print(issubclass(Shape, Rectangle))
```

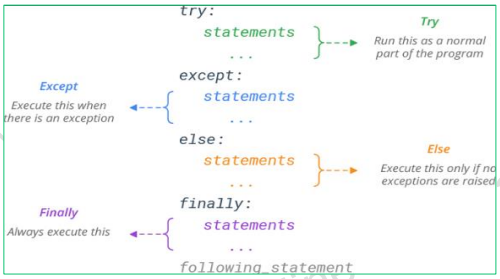

Inheritance – Methods Overriding

- Inheritance of Attributes: The child class inherits all the attributes (data members) of the parent class.
- Inheritance of Methods: The child class inherits all the methods (functions) of the parent class.
- Overriding Methods: The child class can override the methods of the parent class by redefining them.
- Accessing Parent Class Attributes: The child class can access the attributes of the parent class using the ***super()*** function.
- Multiple Inheritance: Python supports multiple inheritance, which allows a class to inherit from multiple parent classes.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def eat(self):
        print(f"{self.name} is eating")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking")

    def eat(self):
        print(f"{self.name} is eating a bone")

dog = Dog("Rex", "Golden Retriever")
dog.eat()
dog.bark()
```

Exceptions

- Exceptions
  - Exceptions refer to the events of errors experienced when executing a program.
    - User Input
    - External resources do not meet requirements
  - Exceptions may cause the Python interpreter to generate an error message and the program to terminate abnormally.
- Handling Exceptions
  - You can handle exceptions using a try-except block. This allows you to catch and handle specific exceptions that may occur in your code.




```
while True :
    try :
        x = int ( input ( "Please enter an integer: " ))
        break
    except ValueError :
        print ( "Oops! That was not valid. Try again..." )
print ( 'Well done!' )
```


Exceptions

- Raising Exceptions:
  - You can raise an exception using the raise statement.
  - This is typically done when a specific condition or error occurs in your code.
- Multiple Exceptions:
  - You can catch multiple types of exceptions using a single except block.
  - Order is important

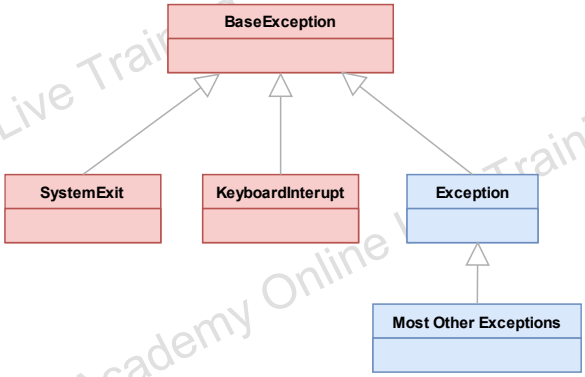
```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

result = divide(10, 2)
print(result)

result = divide(10, 0)
print(result)
```

```
try:
    result = divide(10, "two")
    print(result)
except (ValueError, TypeError) as e:
    print(f"Error: {e}")
```

Exceptions

- Raising Custom Exceptions:
  - You can define your own custom exceptions by subclassing the Exception class.

```
class MyCustomError(Exception):
    def __init__(self, message):
        self.message = message

try:
    raise MyCustomError("This is a custom error")
except MyCustomError as e:
    print(f"Error: {e.message}")
```



Working with Libraries

Python to Advance

Working With Functions Not From You

- Built-in Functions
  - 1. just use it
  - Examples:
    - print
    - input
    - open
    - range
    - str
    - type
    - ...
- Standard Modules
  - 1. import
  - 2. use it
  - Examples:
    - os
    - sys
    - glob
    - math
    - random
    - re
    - ...
- External Libraries
  - 1. install
  - 2. import
  - 3. use it
  - Examples:
    - requests
    - numpy
    - pandas
    - matplotlib
    - scikit-learn
    - ...

Built-in Functions

print(*objects, sep=' ', end='\n', file=None, flush=False)

- Print objects to the text stream file, separated by sep and followed by end. sep, end, file, and flush, if present, must be given as keyword arguments.
- All non-keyword arguments are converted to strings like str() does and written to the stream, separated by sep and followed by end. Both sep and end must be strings; they can also be None, which means to use the default values. If no objects are given, print() will just write end.
- The file argument must be an object with a write(string) method; if it is not present or None, sys.stdout will be used. Since printed arguments are converted to text strings, print() cannot be used with binary mode file objects. For these, use file.write(...) instead.

Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Standard Library - Overview

- Python's standard library is very extensive, offering a wide range of facilities.
- The library contains built-in modules that provide access to system functionality such as file I/O, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming.

- Operating System Interface
- File Wildcards
- Command Line Arguments
- Error Output Redirection and Program Termination
- String Pattern Matching
- Mathematics
- Internet Access
- Dates and Times
- Data Compression

- Performance Measurement
- Quality Control
- Output Formatting
- Templating
- Working with Binary Data Record Layouts
- Multi-threading
- Logging
- Weak References
- Tools for Working with Lists
- ...

Standard Library – Some Popular Modules

- os: Provides a way of using operating system dependent functionality.
- sys: Provides access to some variables used and to functions that interact with the interpreter.
- math: Provides access to mathematical functions.
- random: Provides functions for generating random numbers.
- time: Provides various time-related functions.
- datetime: Provides classes for manipulating dates and times.
- re: Provides support for regular expressions.
- json: Provides functionality for working with JSON data.
- logging: Provides support for logging events and messages.
- csv: CSV file reading and writing.
- ...

Regular Expression

- What is a regular expression
  - It is a powerful weapon for matching strings.
  - Its essence is a descriptive language
    - Define a rule /pattern for a string
    - Any string that meets the rules is considered a "match"
  - Normal and special characters
    - A normal character matches itself
    - Special characters have control functions:
      - `+?.*^$()[]{}|\`
- Application Scenario
  - Verify text / Extract text / Split text / Replace text

Character Classes	describe	model	match
[char_group]	match char_group By default , the match is case sensitive .	[ae]	"a" in "gray" " a" and "e" in "lane"
[^ char_group]	Negation: matches any single character not in character_group. By default, characters in character_group are case sensitive.	[^aei]	The "r", "g", and "n" in "reign"
[first - last]	Character range: matches any single character in the range from first to last.	[AZ]	"A" and "B" in "AB123"
.	Wildcard: Matches any single character except \n. To match a literal period character (. or \u002E), you must precede the character with the escape character (\.).	ae	"ave" in "nave" "ate" in "water"
\w	Matches any word character.	\w	"I", "D", "A", "1", and "3" in "ID A1.3"
\W	Matches any non-word character.	\W	" " and "." in "ID A1.3 "
\s	Matches any whitespace character.	\w\s	"D" in "ID A1.3"
\S	Matches any non-whitespace character.	\s\S	" _" in "int __ctr"
\d	Matches any decimal digit.	\d	"4" in "4 = IV"
\D	Matches any character that is not a decimal digit .	\D	" ", "=", "", "I", and "V" in "4 = IV"

re

- Verify text
  - match() / fullmatch ()
    - Matches a pattern from the beginning of the string
    - If the match is successful, a match object is returned; otherwise, None is returned.
- Extract text
  - search()
    - Scans the entire string and returns the first successful match
    - If the match is successful, a match object is returned; otherwise, None is returned.
  - findall ()
    - Find all substrings that match the regular expression and return them as a list

```
import re

pattern = r"^Hello"

test_strings = [
    "Hello world!",
    "Goodbye world!",
    "Hello, how are you?",
    "Goodbye, see you later!"
]

for test_string in test_strings:
    match = re.match(pattern, test_string)
    full_match = re.fullmatch(pattern, test_string)

    if match:
        print(f"Match found in '{test_string}' using re.match()")
    else:
        print(f"No match found in '{test_string}' using re.match()")

    if full_match:
        print(f"Match found in '{test_string}' using re.fullmatch()")
    else:
        print(f"No match found in '{test_string}' using re.fullmatch()")
```

re

- Split text
  - split()
    - Split a string into a list at regular matches
    - re.split (pattern, string[, maxsplit =0])
    - .split(string[, maxsplit =0])
- Replace text
  - sub()
    - Find all matches of a pattern and replace them with a different string
    - re.sub (pattern, replacement, string, count=0)
    - .sub(replacement, string[, count=0])
  - subn ()
    - Do the same work as sub() , but returns a 2- tuple

```
import re

string = "Hello, how are you?"
pattern = r"[ ,]"

result = re.split(pattern, string)
print(result)
```

```
string = "Hello, how are you?"
pattern = r"[ ,]"
replacement = "_"

result = re.sub(pattern, replacement, string)
print(result)
```

```
string = "Hello, how are you?"
pattern = r"[ ,]"
replacement = "_"

result, count = re.subn(pattern, replacement, string)
print(result)
print(count)
```


Using Regular Expression to Process Data

```
import re

line = 'FDN
:SubNetwork=Python,MeContext=NodeP'
matches = re.match('^FDN : (.*)$', line)
if matches:
    mofdn = matches.group(1)
    print('The MO FDN is: ', mofdn)
```

```
The MO FDN is: SubNetwork = Python,MeContext = NodeP
```

```
import re
line = 'FDN : SubNetwork=P,MeContext=NodeP'
m = re.match('^FDN : (.*,(.*=(.*)))$', line)
if m:
    mofdn = m.group(1)
    moldn = m.group(2)
    moid = m.group(3)
    print('The MO FDN is:', mofdn,
    'and the MO LDN is:', moldn,
    'and the MO ID is:', moid)
```

```
The MO FDN is: SubNetwork = P,MeContext = NodeP
and the MO LDN is: MeContext = NodeP and
the MO ID is: NodeP
```

datetime

- The datetime module provides several classes for working with dates and times.
- datetime class:
  - This is the main class in the datetime module.
  - It represents a specific date and time.
- timedelta class:
  - represents a duration of time.
  - It is used to perform date and time calculations, such as adding or subtracting days, hours, minutes, or seconds.

```
from datetime import datetime, timedelta

# Creating datetime objects
now = datetime.now()
specific_date = datetime(2022, 1, 1, 12, 0, 0)

# Accessing date and time components
year = now.year
month = now.month
day = now.day
hour = now.hour
minute = now.minute
second = now.second

# Formatting the datetime object as a string
formatted_date = now.strftime("%Y-%m-%d")
formatted_time = now.strftime("%H:%M:%S")
formatted_datetime = now.strftime("%Y-%m-%d
%H:%M:%S")

# Performing date and time calculations
one_day_later = now + timedelta(days=1)
one_day_earlier = now - timedelta(days=1)
one_hour_later = now + timedelta(hours=1)
one_hour_earlier = now - timedelta(hours=1)
```


datetime

- date class:
  - This class represents a date without the time component.
- time class:
  - This class represents a time without the date component.
- strftime():
  - This method is used to format a datetime object as a string.
- strptime():
  - This method is used to parse a string and convert it into a datetime object. (datetime only)

```
from datetime import date

today = date.today()
specific_date = date(2022, 1, 1)

year = specific_date.year
month = specific_date.month
day = specific_date.day

formatted_date = specific_date.strftime("%Y-%m-%d")
```

```
from datetime import time

specific_time = time(12, 0, 0)

hour = now_time.hour
minute = now_time.minute
second = now_time.second

formatted_time = now_time.strftime("%H:%M:%S")
```

CSV

- csv module is used for reading and writing CSV (Comma Separated Values) files.
- csv.reader():
  - This function is used to read a CSV file and return a reader object.
  - The reader object can be iterated over to retrieve rows from the CSV file.
- csv.writer():
  - This function is used to write to a CSV file.
  - The writer object can be used to write rows to the CSV file.

```
import csv

with open('example.csv’, 'r', newline='') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

```
data = [
    ['Name', 'Age'],
    ['Alice', 25],
    ['Bob', 30]
]
with open('example.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

```
with open('example.tsv', 'w', newline='') as file:
    writer = csv.writer(file,
dialect=csv.excel_tab)
    writer.writerows(data)
```


CSV

- csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds): Creates a reader object that reads rows from a CSV file and returns each row as a dictionary.
- csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds): Creates a writer object that writes rows to a CSV file using a dictionary.

```
import csv

with open('example.csv', ‘r’, newline='') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row)
```

```
import csv

data = [
    {'Name': 'Alice', 'Age': 25},
    {'Name': 'Bob', 'Age': 30}
]

with open('example.csv', 'w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=data[0].keys())
    writer.writeheader()
    writer.writerows(data)
```

External Libraries - Overview

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the **Python Package Index**.

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

PyPI helps you find and install software developed and shared by the Python community.

https://pypi.org/

External Libraries – Example Packages

numpy: A library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
pandas: A software library written for the Python programming language for data manipulation and analysis.
requests: A simple, yet elegant HTTP library.
matplotlib: A Python 2D plotting library which produces publication quality figures in a variety of formats.
scikit-learn: A machine learning library for Python.
seaborn: A Python visualization library based on matplotlib.
jupyter: An interactive web-based environment for interactive computing.
beautifulsoup4: A Python library designed for pulling data out of HTML and XML files.
opencv-python: A library of Python bindings designed to solve computer vision problems.
pytest: A mature full-featured Python testing tool.

Package Management

- pip is a package manager for Python. It is used to install, update, and manage Python packages and dependencies.
- Installing Packages: You can use pip to install packages from the Python Package Index (PyPI) or from a local file.
- Updating Packages: You can use pip to update packages to their latest versions.
- Uninstalling Packages: You can use pip to uninstall packages.
- Listing Installed Packages: You can use pip to list the installed packages and their versions.

```
# install the latest version
pip install package_name

# install the specific version
pip install package_name == version

# install from other indexes
pip install -i http://my.repo/simple package_name
```

```
# upgrade to the latest version
pip install --upgrade package_name
```

```
# uninstall a package
pip uninstall package_name
```

```
# list installed packages
pip list
```


Package Management

- Requirements
  - pip install –r requirements.txt
  - The command is used to install the Python packages listed in a requirements.txt file.

```
# install all required packages in one go
pip install -r requirements.txt
```

```
# example of requirements.txt
requests==2.25.1
numpy==1.20.3
```

Virtual Environment

- In Python, a virtual environment is an isolated environment that allows you to manage dependencies and packages for a specific project.
- Virtual environments are useful for several reasons:
  - Isolation: Each virtual environment has its own set of installed packages and dependencies. This allows you to have different versions of packages for different projects without conflicts.
  - Consistency: Virtual environments ensure that the same code runs consistently across different machines and environments.
  - Reproducibility: By capturing the dependencies and packages specific to a project, you can easily reproduce the environment on another machine.
  - Security: Virtual environments help to prevent conflicts and security vulnerabilities between different projects.

Virtual Environment

- Creating a Virtual Environment:
  - You can create a virtual environment using the ***venv*** module that comes with Python.
- Activating a Virtual Environment:
  - To activate the virtual environment, the steps depend on the operating system.
  - Once the virtual environment is activated, the command prompt or terminal will indicate the name of the active virtual environment.
- Deactivating a Virtual Environment:
  - You can simply close the terminal or command prompt. Alternatively, you can use the ***deactivate*** command.

```
# Create a virtual environment

python -m venv myenv
```

```
# Activate a virtual environment
#On Windows
myenv\Scripts\activate

# macOS and Linux
source myenv/bin/activate
```

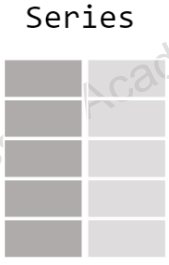
```
# Install packages
Pip install package_name

# Run python code
Python my_script.py
```

Pandas - Overview

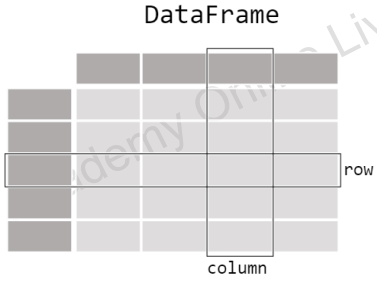
- Pandas is a powerful and versatile library for working with structured data in Python. It simplifies data manipulation, analysis, and visualization tasks, making it a popular choice for data scientists and analysts.
  - DataFrames: Data structure, which is similar to a table in a relational database. DataFrames allow you to store and manipulate tabular data, with rows representing observations and columns representing variables.
  - Data Loading and Exporting: Pandas provides various functions to load data from different sources, such as CSV, Excel, SQL databases, and more. It also supports exporting data to different formats.
  - Data Manipulation: Pandas offers a wide range of functions for data manipulation, such as filtering, sorting, grouping, merging, and reshaping data. Data Cleaning and Preprocessing: Pandas provides tools for handling missing values, data cleaning, and data preprocessing.
  - Data Analysis: Pandas provides a variety of statistical functions for data analysis, such as calculating summary statistics, correlations, and performing statistical tests.
  - Data Visualization: Pandas integrates well with popular data visualization libraries in Python, such as Matplotlib and Seaborn, allowing you to create various types of visualizations.

Pandas - Data Structure

- **Series**:
  - A Series is a one-dimensional labeled array capable of holding any data type.
  - It is similar to a list or array, but with labeled indices.
  - Each element in a Series can be accessed using its index.
  - Series can be created from various sources, such as lists, arrays, or even from a single value.



- **DataFrame**:
  - A DataFrame is a two-dimensional labeled data structure, similar to a table in a relational database.
  - It is capable of holding data of different data types.
  - Each row is called a record, each column is called a field.
  - DataFrames can be created from various sources, such as CSV files, SQL DB, or even from a list of dictionaries.



Pandas - Data Structure

- **Series Object**:
  - Python dict, list / numpy array / ...

```
In [7]: d = {"b": 1, "a": 0, "c": 2}

In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
```

- **DataFrame object**:
  - Python dict, list / numpy 2D array / ...
  - Series objects

```
In [44]: d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=["a", "b", "c", "d"])
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```


Pandas - Data I/O

- Pandas data I/O API provides series of reader and writer

Writing to a csv file:

```
In [142]: df.to_csv("foo.csv")
```

Reading from a csv file:

```
In [143]: pd.read_csv("foo.csv")
Out[143]:
     Unnamed: 0          A          B          C          D
0    2000-01-01   0.350262   0.843315   1.798556   0.782234
1    2000-01-02  -0.586873   0.034907   1.923792  -0.562651
2    2000-01-03  -1.245477  -0.963406   2.269575  -1.612566
3    2000-01-04  -0.252830  -0.498066   3.176886  -1.275581
4    2000-01-05  -1.044057   0.118042   2.768571   0.386039
..          ...        ...        ...        ...        ...
995  2002-09-22 -48.017654  31.474551  69.146374 -47.541670
996  2002-09-23 -47.207912  32.627390  68.505254 -48.828331
997  2002-09-24 -48.907133  31.990402  67.310924 -49.391051
998  2002-09-25 -50.146062  33.716770  67.717434 -49.037577
999  2002-09-26 -49.724318  33.479952  68.108014 -48.822030

[1000 rows x 5 columns]
```

Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Stata	read_stata	to_stata
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

Pandas - Data Viewing

- Quickly preview
  - df.head(n)
  - df.tail(n)
- View the basic information of the data
  - df.size / df.shape
  - df.info()
  - df.index / df.columns
- Quickly view a statistical summary of your data
  - df.describe()
- Sort
  - index：df.sort_index(axis=1, ascending=False)
  - value：df.sort_values(by=label)

```
In [19]: df.describe()
Out[19]:
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.073711	-0.431125	-0.687758	-0.233103
std	0.843157	0.922818	0.779887	0.973118
min	-0.861849	-2.104569	-1.509059	-1.135632
25%	-0.611510	-0.600794	-1.368714	-1.076610
50%	0.022070	-0.228039	-0.767252	-0.386188
75%	0.658444	0.041933	-0.034326	0.461706
max	1.212112	0.567020	0.276232	1.071804

```
In [22]: df.sort_values(by='B')
Out[22]:
```

	A	B	C	D
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-06	-0.673690	0.113648	-1.478427	0.524988
2013-01-05	-0.424972	0.567020	0.276232	-1.087401

Pandas - Data Selection

- Column
  - single：df['column'] / df.colunm
  - multiple：df[['col1', 'col2']]
- Row (slicing)
  - df[0:3]
  - df['2022-02-22': '2022-02-24']
- Labels
  - df.loc['rows', 'columns']
  - df.loc[: , :]
  - df.loc[: , ['A', 'C']]
- Position
  - df.iloc[rows, columns]
  - df.iloc[: , :]
  - df.iloc[: , [0,2]]

```
In [54]: df[0:3]
Out[54]:
```

	A	B	C	D
2022-02-22	-1.406789	0.682477	1.622777	0.443227
2022-02-23	0.345644	-0.154074	-2.140273	0.489833
2022-02-24	-0.068854	1.766517	-1.617096	-0.906261

```
In [53]: df.loc['2022-02-22': '2022-02-24', ['A', 'B']]
Out[53]:
```

	A	B
2022-02-22	-1.406789	0.682477
2022-02-23	0.345644	-0.154074
2022-02-24	-0.068854	1.766517

```
In [55]: df.iloc[1:3, :]
Out[55]:
```

	A	B	C	D
2022-02-23	0.345644	-0.154074	-2.140273	0.489833
2022-02-24	-0.068854	1.766517	-1.617096	-0.906261

Pandas - Data Cleaning

❖ Missing Data
- What are NaN, NA or NaT
- Find missing data:
  - df.isna() / df.isnull()
  - df.notna() / df.notnull()
  - df.info()
- Drop missing data: df.dropna()
  - Optional: axis=0('index') / 1('columns')
  - Optional: inplace=True
  - Optional: how='any' / 'all'
- Fill missing data: df.fillna(value)
  - Mandatory: value=<value>
  - Optional: axis=0('index') / 1('columns')
  - Optional: inplace=True
- Fill missing data: df.ffill() / df.bfill()
- Fill missing data: df.interpolate()

```
In [8]: ser = pd.Series([pd.Timestamp("2020-01-01"), pd.NaT])

In [9]: ser
Out[9]:
0   2020-01-01
1          NaT
dtype: datetime64[ns]

In [10]: pd.isna(ser)
Out[10]:
0    False
1     True
dtype: bool
```

```
In [84]: df
Out[84]:
     0  1    2
0  NaN  1  2.0
1  1.0  2  NaN
2  1.0  2  3.0

In [85]: df.dropna()
Out[85]:
     0  1    2
2  1.0  2  3.0

In [86]: df.dropna(axis=1)
Out[86]:
   1
0  1
1  2
2  2
```

```
In [91]: df
Out[91]:
    np  arrow
0  1.0    1.0
1  NaN   <NA>
2  NaN   <NA>
3  2.0    2.0

In [92]: df.fillna(0)
Out[92]:
    np  arrow
0  1.0    1.0
1  0.0    0.0
2  0.0    0.0
3  2.0    2.0
```


Pandas - Data Cleaning

- Duplicated Data
  - Find duplicated: df.duplicated()
  - Drop duplicated: df.drop_duplicates()
    - Optional: subset=<column label or labels>
    - Optional: inplace=True
    - Optional: keep='first' / 'last' / False
      - 'first': Discard duplicate data except for the first occurrence (default)
      - 'last': Discard duplicate data except for the last occurrence
      - 'False': Discard all duplicate data

```
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

To remove duplicates on specific column(s), use `subset`.

```
>>> df.drop_duplicates(subset=['brand'])
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
```

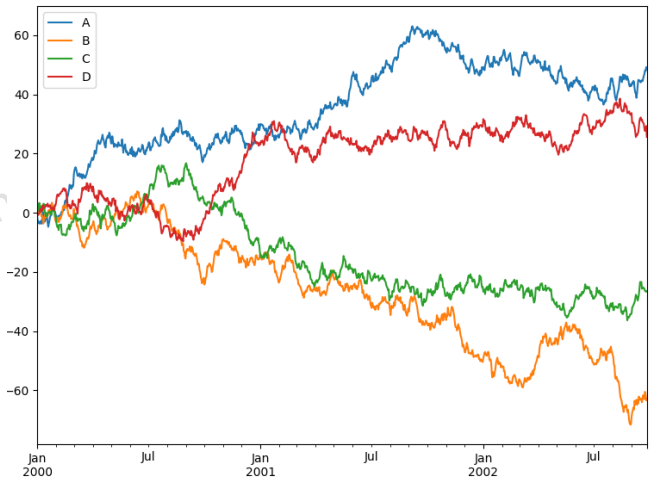
To remove duplicates and keep last occurrences, use `keep`.

```
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
    brand style  rating
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
4  Indomie  pack     5.0
```

Pandas - Data Visulization

- Use matplotlib's API for visualization
  - import matplotlib.pyplot as plt
  - plot() method of data to call plt.plot()

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

ts = pd.Series(np.random.randn(1000),
        index=pd.date_range("1/1/2000",
        periods=1000))

df = pd.DataFrame(np.random.randn(1000, 4),
        index=ts.index,
        columns=["A", "B", "C", "D"])

df = df.cumsum()

df.plot()
plt.show()
```



Course Summary

In this course, we discussed how to:

- Using data structures more flexibly
- Writing more complexed functions
- Practicing Object-Oriented Programming
- Working with python libraries