CA341 Comparative Programming Languages

Assignment 3 – Group Assignment

Analytical Analysis of a Programming Language (Python)

[redacted]

Table of Contents

Declaration on Plagiarism

Assignment Submission Form

This form must be filled in and completed by the student(s) submitting an assignment

Name(s): David Weir, [redacted]
Programme: Comparative Programming Languages
Module Code: CA341
Assignment Title: Analytical Analysis of a Programming Language
Submission Date: 05/12/2021
Module Coordinator: Dr. Brian Davis

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at
http://www.dcu.ie/info/regulations/plagiarism.shtml ,
https://www4.dcu.ie/students/az/plagiarism

and/or recommended in the assignment guidelines.

Name(s): David Weir, [redacted]

Date: 05/12/2021

Introduction

This is our critical analysis of the Python programming language under the statement “Jack of all trades, master of none!”. The meaning of this of course being that Python’s versatility in many areas is, in fact, a weakness as it does not provide adequate support for anything.

We believe that although Python is versatile and can be described as a “Jack of all trades”, this is not a weakness of the language. No language is perfect and all have their own pros and cons but it would be a mistake to say Python is a “master of none”. Python is extremely versatile and while it does try to do many things, it does not implement them poorly.

We will be performing an in-depth analysis of Python in 5 key areas to support our viewpoint. These areas are:

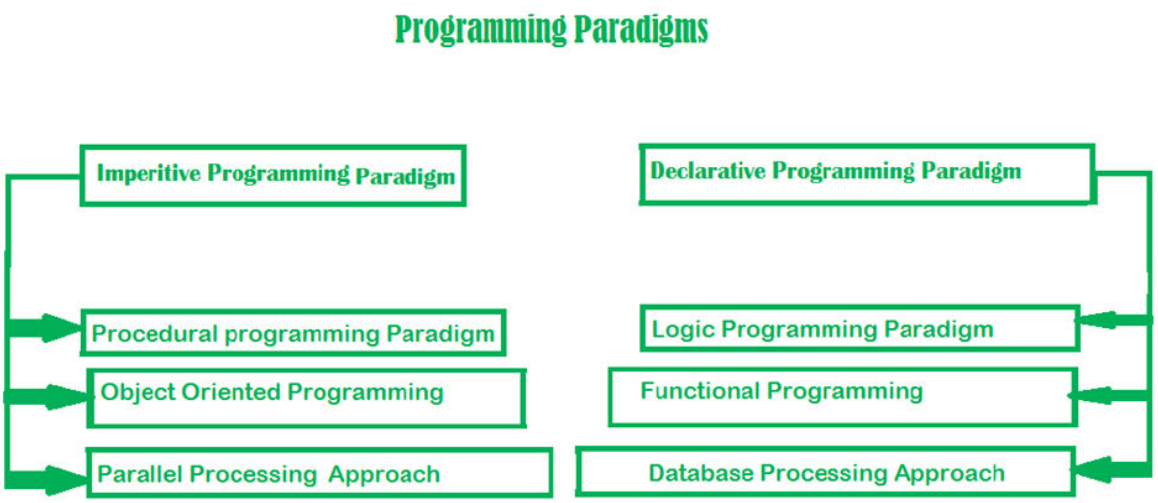
- Data Types in Python
- Analysis of Python’s programming paradigm supports
- Abstraction in Python
- Object-Oriented Programming support
- Memory Management

Team Breakdown

Team Member	Role(s)
David Weir	Lead writer, Analysis of Python’s programming paradigm supports
[redacted]	Team leader, Object-Oriented Programming support
[redacted]	Data Types in Python, code peer review
[redacted]	Abstraction in Python, peer review
[redacted]	Memory Management, peer review

Analysis of Python's programming paradigm supports

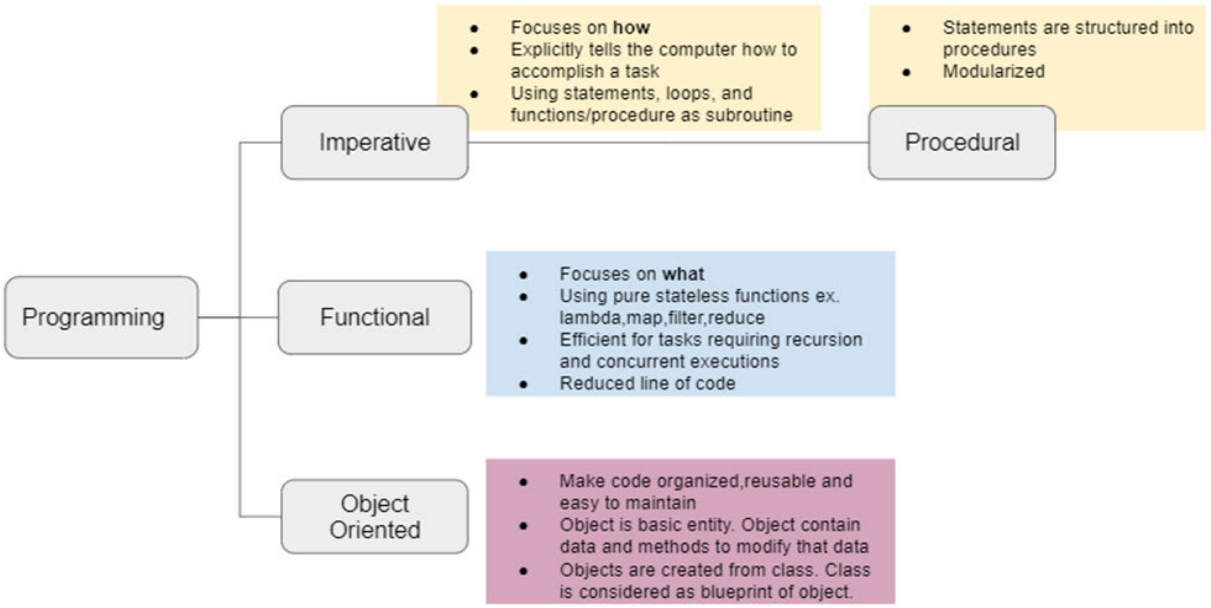
Programming paradigms are used to classify programming languages based on their supported features. That is, paradigms describe a particular style of programming each with their own approaches, tools, logic and features. These paradigms could be thought of as a particular approach to answering a question or solving a problem.[4] The image below is a basic breakdown of some styles of programming.



A basic breakdown of some different programming styles.

Python itself, however, supports multiple programming paradigms (which is becoming normal for most languages). Python supports 3 major programming paradigms:

- Functional
- Object Oriented
- Imperative and by extension Procedural programming paradigms.[1]



Each paradigm has its own strengths and weaknesses and are suited towards certain jobs. That is the beauty of Python's multi-paradigm design, a developer can choose the best tool for the job without needing to learn a new language and Python supports these paradigms fully. We will discuss further the positives of the multi-paradigm approach but first let's look at how Python can be used to implement our 3 programming paradigms.

Object Oriented Programming

The OO paradigm structures its programs into classes and objects as opposed to functions or logic. These classes are a user-defined collection of objects and contain the blueprint to create objects and the class' attributes and function methods.

The OO paradigm has 3 pillars that define it and Python fully supports all 3 pillars of OOP: encapsulation, inheritance and polymorphism.

Classes

First we can see an example of Python using classes. The Dog class has 2 instance attributes, its name and the breed and 2 class methods breed and speak. The attributes and methods belong to the Dog class and can only be called by a Dog object (dog1 and dog2). We can also see a 2nd class Cat with 1 attribute, name, and 1 method, meow. Again this method can only be accessed by a Cat object.

```
class Dog:

    # Instance attribute
    def __init__(self, name, breeds):
        self.name = name
        self.breeds = breeds

    # class methods
    def breed(self):
        print("{} is a {}".format(self.name, self.breeds))

    def speak(self):
        print("My name is {}".format(self.name))

class Cat:
    def __init__(self, name):
        self.name = name

    def meow(self):
        print("{}: Meow".format(self.name))

# Driver code
```

```
# Object instantiation
dog1 = Dog("Luca", "Golden Retriever")
dog2 = Dog("Skye", "Husky")
cat = Cat("Salem")

# Accessing class methods
dog1.speak()
dog2.speak()

dog1.breed()
dog2.breed()

cat.meow()
```

Inheritance

Python also supports the OOP concept of Inheritance. Inheritance is the process of a “child” class taking on the attributes and methods of a “parent” class, the child class is derived / inherits the parent class. The child classes can override or extend the attributes and methods of its parents.[3]

In the code snippet below we can see a quick example of a Python programme implementation using inheritance. In this programme we have a Person class (the parent) and an Employee class (the child) that inherits from the Person class. We create an Employee object and as we can see this object has access to the methods of the parent class, seen in the display function. However, we also see the Employee class’ details function extends the details function inherited from Person. In this way, the child modifies the behaviour of the parent.

```
# Python code to demonstrate inheritence.

# parent class
class Person(object):

    # constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print("Name: " + self.name)
        print("ID: {}\n".format(self.idnumber))
```

```
    def details(self):
        print("My name is {}".format(self.name))
        print("ID: {}".format(self.idnumber))

# child class
class Employee(Person):

    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("Employee Name: {}".format(self.name))
        print("ID: {}".format(self.idnumber))
        print("Salary: {}".format(self.salary))
        print("Position: {}".format(self.post))


# creation of an object variable or an instance
emp = Employee("Daniel", 193762, 200000, "Intern")

# calling a function of the class Person using
# its instance
emp.display()
emp.details()
```

Polymorphism

The 2nd OOP pillar is Polymorphism. The word polymorphism means to have many forms. In our code this means functions with the same name but used for different types. One of Python's greatest strengths in OOP is its dynamic typing which makes it very easy to implement polymorphism. [2]

In our code below we implement a simple example in Python using both Inheritance and Polymorphism. The country classes all inherit from the EU class, so they all have access to the intro function as seen when called with the Ireland object. Each country class also inherits the ismember() function from the parent class. However, this time we use polymorphism to override the parent class' function with the child class' function. We see here there are many different forms of the ismember() function so therefore it is polymorphic.

```
class EU:
  def intro(self):
    print("The EU is a political and economic union with 27 member
states.")

  def ismember(self):
    print("Most countries in Europe are in the EU.\n")

class Ireland(EU):
  def ismember(self):
    print("Ireland is a member of the EU.\n")

class UK(EU):
  def ismember(self):
    print("The UK has left the EU after Brexit.\n")

class Germany(EU):
    def ismember(self):
        print("Germany is in the EU.")

eu = EU()
ire = Ireland()
uk = UK()
ger = Germany()

eu.intro()
eu.ismember()

ire.intro()
ire.ismember()

uk.ismember()
ger.ismember()
```

Encapsulation

Lastly, we have Encapsulation, the concept of wrapping data and the methods that access that data into one unit. This restricts access to variables and methods directly adding a layer of security and preventing the accidental modification of data. Instead, an object's variables can only be accessed by a method under that object. These variables are known as private variables. A class is the most obvious example of encapsulation as it encapsulates its member functions, variables and attributes etc.

Below is an example of encapsulation in Python. Python does not provide syntactic declaration of visibility (public, private, protected) or access control. This can be seen as a downside to Python's support for the OO paradigm however, it does have informal support for encapsulation.[2] The private variable is denoted by "__". In this case we have the private variable attribute self.__priv in the Base class. In the commented code at the bottom we see 2 examples where we try to access this private information only to be met with an Attribute error.

```
# Creating a Base class
class Base:

    def __init__(self):
        self.public = "Some publicly available info"
        self.__priv = "Some private info"

# Creating a derived class
class Inherited(Base):

    def __init__(self):

        # Calling constructor of Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__priv)

# Driver code
object = Base()
print(object.public)

# print(obj1.c) will raise an AttributeError
# AttributeError: 'Base' object has no attribute 'priv'

# object2 = Derived()
# this will also raise an AtrributeError as private member of base
# class is called inside derived class
# AttributeError: 'Derived' object has no attribute '_Derived__priv'
```

Imperative / Procedural Programming

The Procedural programming paradigm is a type of imperative programming that focuses on a series of computations or instructions to be executed. Therefore, we know Python fully supports both paradigms, but it excels at procedural programming. Its programs are built around executing this sequence of instructions and these instructions are split into sets of

instructions called procedures comparable to functions. A procedure's data is localized and cannot be accessed from outside its scope. These procedures are executed serially. This paradigm assists the modularity of code and modularization which is done by the functional implementation.[5]

We can see a procedural implementation of summing the elements of a list in Python below. This program uses Python functions to support modularization. We see in the for loop of the sumlist function we are telling the computer how to sum the list (by adding each value to a final result variable).

```
# Procedural way of finding sum of a list

mylist = [10, 20, 30, 40]

# modularization is done by functional approach
def sumlist(mylist):
    res = 0

    for val in mylist:
        res += val

    return res

print(sumlist(mylist))
```

Functional Programming

The final paradigm supported by python is the Functional paradigm which is based on pure mathematical functions that use expression and recursion to perform computations. It supports higher-order functions and lazy evaluation. It is a declarative paradigm as it uses declarations not overstatements. It is said to focus on “what to solve” and not “how to solve” it, we can see how this contrasts with procedural programming in the image above.

Python is categorized as an Impure Functional Language as it does not solely support the functional paradigm and it is possible to maintain state and create side effects.[6] However, it still supports the Functional Paradigm regardless.

Below we can see the same programme to sum a list but this time it is implemented in Python using the Functional Paradigm. There are 2 ways to do this functionally:

Using a local function, here the code indexes the list to tell the computer what to solve and uses recursion to call the function on itself again until it is only a list of 1 element.

Or using a lambda expression and the functools library for access to higher-order functions.[7]

```
# Functional way of finding sum of a list
import functools


mylist = [11, 22, 33, 44]

# Recursive Functional approach
def sumlist(mylist):

    if len(mylist) == 1:
        return mylist[0]
    else:
        return mylist[0] + sumlist(mylist[1:])

print(sumlist(mylist))

# lambda function approach
print(functools.reduce(lambda x, y: x + y, mylist))
```

How this works in Python's favour

In our analysis of Python and it's supported programming paradigms we have seen both where it excels and where it falls short. Python implements an impure functional paradigm and it does not fully support the OO paradigm either due to its failure to implement data encapsulation, an integral concept of Object-Oriented programming.[7]

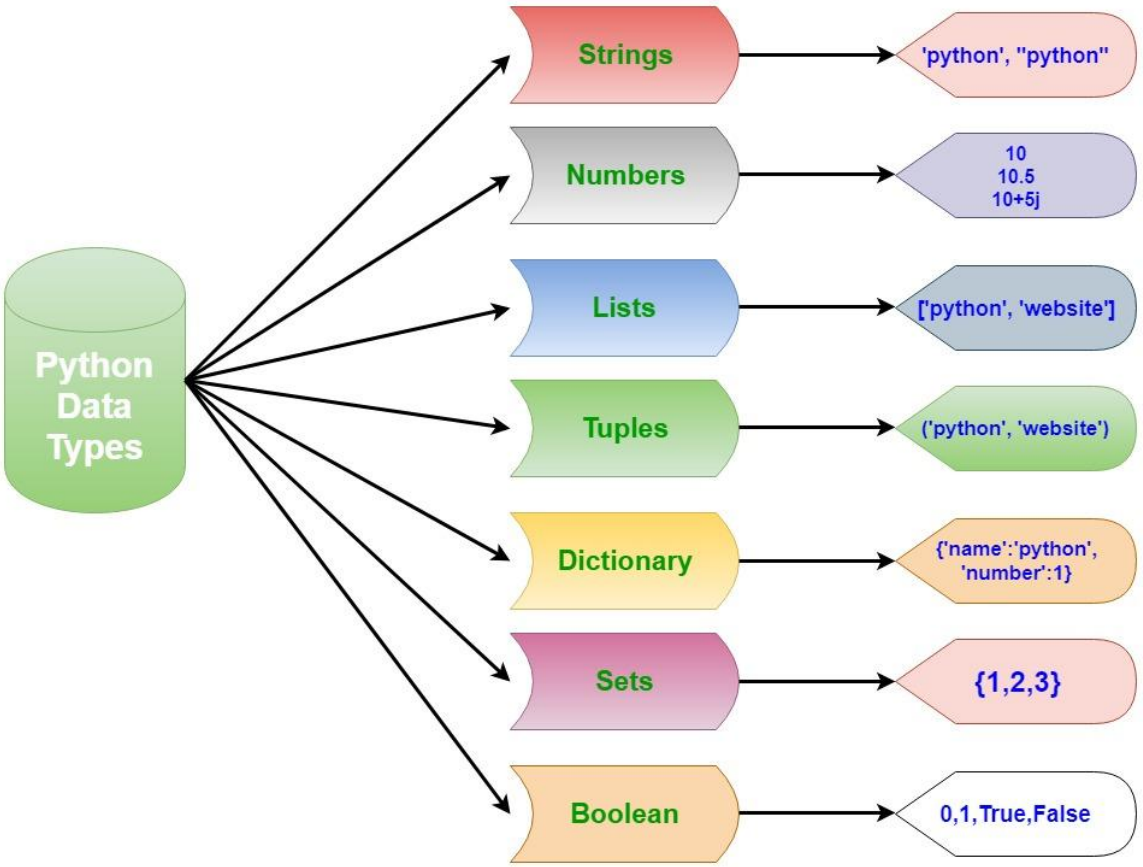
However, it does excel in its support for imperative and procedural programming, implementing these paradigms fully without fault. Despite its fall down in OOP it still supports inheritance, polymorphism, refactorization, classes and all other OO concepts. It even, as we discussed, supports encapsulation and private variables. That is the beauty of Python, although it does not always fully support a paradigm naturally, its extensive libraries and workarounds pick it up where it falls down.

Python's inherent flexibility is its primary strength. It does not tie you down or enforce one particular code style or paradigm, allowing developers to choose the right paradigm and right tools for the job. Python trusts our ability to choose the correct path and gives us the tools to succeed. This is why, on the topic of programming paradigms, Python may be a "Jack of all trades", but it is not a "master of none", in fact it could be argued that it is a "Jack of all trades and a master of some".

Data Types & Scope

Primitive Types

The primitive types in Python are Boolean, Integer, Float and String. In Python a boolean value is either True or False or True can be any non-Zero number and 0 can be treated in comparison as the boolean value False. This is done in many other languages such as C. Integers in Python are defined as any whole number and have a lot of in-built functions which can be used to change the value assigned such as getting the absolute value of the number or changing it into a string for easier manipulation. Floats are the other number type and it covers any of the Z subset of the real numbers. There are functions to turn your number value from an integer to a float and can be very useful. Last but not least is the String type which is a collection of characters. In most other languages there is the character data type for a single character but in Python there is no such thing and only the type String exists. Strings are created using single, double or triple quotation marks in contrast to others like the C related languages which like to have only one of the three. Contents of a string can be indexed and the string once it has been created is immutable.[16]



Composite Types

Python's composite data types are known as lists, tuples and dictionaries. Lists in python are a collection of data that is ordered, mutable and allows duplicates. They can be assigned and called using square brackets. They are iterable so you can use Python's for and while loops to go through its contents and check its contents using the in operator. A great part about Python's list is that they do not have a set length compared to Java's base arrays which are a fixed length upon creation. Tuples in Python are very like Lists but the biggest difference is that they are immutable and so much less versatile but a handy way to collect data. Python's dictionaries are its mapping type. It is unordered and you need a key to access its value, it is based around hashmaps that Java has implemented but lacks some of its methods.[17]

```
#list
a = [1,2,3,4,5]

#tuple
b = (1,2,3,4,5)
```

Dynamically Typed

Python is dynamically typed which means that types do not have to be defined in code and are defined at run-time, which I think lets python code be typed a lot more flexibly, even though this may be a slower method I think this is a great strength of Python. It allows for a lot less code to be written overall and means that variables can be reused and given a different type which stops the user starting to use variable names which may make the code unreadable.

```
#dynamically typed
a = 10
b = "Hello World!"
c = True
```

Scope

As many modern programming languages do these days, Python defines its scope in blocks but it does not use brackets to do it but uses functions, modules and class definitions to show each block's scope. It does this by using a special dictionary mapping called namespaces. Accessing other local variables or other local functions means you need to use dot notation as they will be locally locked to their block.[18]

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

for x in a:
   if x % 2 == 0:
       print(x)
```

Why all this is to Python's advantage

All that I have been shown by Python's data type implementation shows I think that Python is a user-first language, it has seen other languages and has gotten rid of data types the designers thought of as not needed such as the array or the character type and made their superior types lists and strings the norm. Lists I think are its greatest advantage as not needing to define the length at creation saves a lot of time and some awkward computation.

Pythons use of different typing system and Scope definition sets it apart from most OO programming languages and it really makes you learn to properly modulate your code and indent properly, though this can be a downfall for some beginner programmers as it is a lot easier to see a bracket is out of place than a tab or space out of place.

Abstraction in Python

What is abstraction?

Abstraction is [12] that feature in OOP concept wherein the user is kept unaware of the basic implementation of a function property. The user is only able to view basic functionalities whereas the internal details are hidden.
It's really a simple concept so I will give an example of abstraction in the real world: for someone to drive a car they do not need to understand the internal functioning and complexity of the car like how the engine works etc. They just need to know how to use the external objects such as the pedals, steering wheel, handbrake etc. and see it as a well-defined object with its own unique behaviour.

What are abstract classes and methods?

Abstract classes and methods are key in object-oriented programming.
[13]An abstract class can be considered as a blueprint for creating other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. An abstract class is a class containing one or many abstract methods. An abstract method refers to a method that has a declaration but does not have an implementation.

How is abstraction implemented in Python?

In python you must import the module ABC(Abstract Base Classes)and import the abstract method if you want to implement an abstract class, this is because Python cannot provide abstract classes itself. ABC allows us to decorate methods and make them abstract using the keyword @abstractmethod.
For example:

```
from abc import ABC, abstractmethod
```

Using this decorator shows that the following sleep method is abstract.

For example:

```
class Dog(ABC):
   @abstractmethod
   def sleep(self):
       pass
```

However, in Java I find that abstract classes are structured better as seen below. For example, we needed to use the abstract keyword when defining the class. We do the same thing when applying the abstract method “sleep”, firstly the abstract keyword is used followed by the return type and method name.

```
public abstract class Dog{
   public abstract void
sleep();
}
```

This structure is better than Python because it’s easier to read and there was no need to import any libraries like in Python.

Abstract methods in Python can also have an implementation unlike in Java. Once subclasses use this method, it must be overridden. For example:

```
from abc import ABC, abstractmethod
class Dog(ABC):

   @abstractmethod
   def sleep(self):
       print("sleeping")

class Bulldog(Dog):
   def sleep(self):
       super().sleep()
       print("bulldogs love sleep”)
```

In this example, the Dog class has an abstract method ‘sleep’. Within this method, there is a body. This implementation gets overridden when the Bulldog class inherits this method. Java has more strict abstract methods as it does not allow implementation in them and produces an error message ‘error: abstract methods cannot have a body’. Abstract methods are supposed to be templates but Python doesn't hold up to the same standard. By allowing someone to provide implementation, it defeats the original purpose of creating the abstract method.

[14]Abstraction has many advantages in that it can reduce programming efforts and reduce the complexity of the code. It allows us to hide unwanted details from the user and therefore increases the security of a program. It also allows focusing on the main concept. It prevents duplication of code in terms of using code that shares the same data type, we could just use

an abstract class and put the code there. It allows for code reusability. It prevents the user from writing low level code. We can change a program without affecting the user as the implementations would all be internal which is very beneficial.

[15]In conclusion, we can see how while Python can handle abstraction well to some extent, java provides a better structure, is more efficient and stays true to what abstraction really is. This links back to the statement “jack of all trades, master of none!”, although this statement may apply to Python in terms of abstraction it does not apply to Python in terms of other OOP concepts discussed elsewhere in this analytical analysis. Abstraction also does not mean to avoid what isn't necessary for a specific operation. It is good practice for a programmer to define a separate method in which you can store less important data so that it can be used later if needed.

Object-Oriented Programming support

As stated before python is heavily geared towards Object Oriented Programming. In this section we are going to compare how python fairs against a programming language that is designed from the ground up with Object Oriented in mind (Java). All the examples here will be the same as above but in Java instead.

Classes

To begin with classes consisting of instances and methods, in Java this remains mostly the same apart from a couple minor differences which will be touched upon later.

```java
public class Dog {
    //instance variables
    String name;
    String breed;

    //constructor
    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }

    //methods
    public String breed() {
        return breed;
    }
```

```
    public String speak() {
        return "Woof";
    }

public class Cat {
    //instance variables
    String name;;

    //constructor
    public Cat(String name) {
        this.name = name;
    }

    //methods
    public String meow() {
        return "Meow";
    }
}
//main method
public static void main(String[] args) {
    //object instantiation
    dog dog1 = new dog("Luca", "Golden Retriever");
    dog dog2 = new dog("Skye", "Husky");
    cat cat1 = new cat("Salem");

    //Accessing class methods
    dog1.speak();
    dog2.speak();

    dog1.breed();
    dog2.breed();

    cat1.meow();
}
```

As we can see it takes substantially longer to do the same thing in Java than it does to do in python, this is mainly because of the inclusion of having to make instance variables as well as a constructor first, in python this is done through the encapsulation where the variables are passed through the “__init__” function. This is an improvement from the Java way of making classes as it allows for easier reading of each class when it's broken down to just function definitions.

Inheritance

In Java the class definitions would often be in separate files with the same name as the class, this allows for easy and clear inheritance between classes. In Java it would look like this.

```
public class Person {
   private String name;
   private int idnumber;

   //constructor
   public Person(String name, int idnumber) {
       this.name = name;
       this.idnumber = idnumber;
   }

   public display() {
       System.out.println("Name: " + name);
       System.out.println("ID: " + idnumber);
   }
}

public class Employee extends Person {
   private float salary;
   private String post;

   //inherited constructor
   public Employee(String name, int idnumber, float salary, String
post) {
       super(name, idnumber);
       this.salary = salary;
       this.post = post;
   }

   public details() {
       System.out.println("Employee Name: " + name);
       System.out.println("Employee ID: " + idnumber);
       System.out.println("Salary: " + salary);
       System.out.println("Post: " + post);
   }
}

pubic public static void main(String[] args) {
   emp = new Employee("John", 12345, 50000, "Manager");
```

```
    emp.display();
    emp.details();
}
```

In Python when inheriting you are able to inherit from multiple classes at the same time. This can be advantageous as Java does not allow for a class to inherit from more than one parent at the same time, can see this with the “extends” keyword when defining the class.

Polymorphism

Python is dynamically typed and is able to pick up on the context of how it's being used. This is vastly simpler than it is in Java because in order to override methods in Java you would have to use an interface as follows.

```
//interface
public interface EU {
    public void isMember();
}

public class Ireland implements EU {
    public void isMember() {
        System.out.println("Ireland is a member of EU");
    }
}
```

This works similarly to how python can inherit multiple classes, in Java multiple classes can inherit multiple interfaces. In python to override a function with the same name all that has to be done to make it polymorphic is changing how the function runs when you initialise it which becomes quite intuitive.

Encapsulation

Being one of the most fundamental concepts of Object Oriented Programming, this is one of the areas where python falls short when compared to other Object Oriented programming languages. In languages like Java or C++ you can declare a variable, method or even a class public, private or protected. Which can be seen in the code examples above.

Protected classes in python (denoted by double underscores on both side of the name, sometimes called dunder classes) such as __init__ cannot be changed from within another class but it can be overridden inside of another class, an example of which would be the __str__ class, normally when printing the class it just returns the memory address that can be overridden by calling it.

Private variables and functions do exist; they are just denoted with a double underscore in front of the name examples of which can be seen above. By default everything is public in python.

The implementation of wrapping data under one sheath in python, while it can be done it is counter intuitive, and when thinking back to the phrase “Jack of all trades master of none” this is where it falls down short when thinking about Object Oriented support of a language. This downfall can have its implications when it comes to security when data isn’t wrapped properly.

Conclusion

The conclusion for Pythons implementation of Object Oriented support is that it implements the fundamentals of Object Oriented quite well but it falls down in certain areas mainly in its implementation of encapsulation but at the same time it has its advantages especially if you look at inheritance it becomes easier almost to inherit from multiple classes. Because python is spreading itself over many different paradigms of programming it truly embraces the phrase “jack of all trades, master of none”.

Memory Management

Memory management in python is [8] the process whereby efficient allocating, de-allocating and coordinating of memory is done, allowing all the various processes to run in a smooth manner and access various system resources optimally. The cleaning of memory objects that aren't being acquired any longer is also a part of memory management.

The memory manager in python is in charge of these sorts of tasks by allocating, managing and periodically running to clean up the memory. In Python, objects are managed by reference counting, dissimilar to other programming languages. Reference counting is a simple procedure where objects are deallocated when they are not referenced in the program. Each variable in python is a reference/ pointer not to the genuine value itself but to an object. Meaning that the memory manager keeps a record of the number of references to each object in the program. [9] The garbage collector (a part of the memory manager) automatically releases memory from a specific object if that object's reference count falls to zero, meaning that the object is no longer in use. The procedure of allocating and deallocating memory is automatic so a user would not have to worry about this, the recovered memory can be utilised by other objects.

In Python, there are two types of memory:
1. Heap - Dynamic Memory
2. Stack - Static Memory

All the values objects are contained in a private heap and all the references and methods are contained in stack memory.

1. Dynamic Memory

Heap allocation stores dynamic memory, meaning that memory can be freed and recovered when it's no longer being used. The memory is allocated at runtime. In python, everything is an object which is very practical as it allows the dynamic memory allocation to underlie python memory management. The python memory manager will automatically recover memory from objects that are no longer in use.

```
#variable allocated in a heap
d = [1, 2, 3] * 5
```

2. Static memory

Stack allocation stores static memory. The memory is allocated at compile time. Stack is used for applying static allocation, meaning memory cannot be reused.

```
#each variable assigned to a stack
a = 10
b = {1: 'a', 2: 'b'}
c = [1, 2, 3]
```

Python executes a process known as ‘interning’ to optimize memory allocation. Few objects in python only contain one object in dynamic memory while calling on various other variables to spot the memory address if those objects are being utilized. The objects that python performs ‘interning’ on are boolean, integer number [-5, 256] and some strings. Other types of objects such as most strings, floars, lists, dictionaries, tuples and large integers interning does not apply to these.

The Garbage Collection is the process where python [10] releases blocks of memory which are no longer in use. Objects that are not references in the program are removed by Python to free up memory space. The garbage collector is run during the execution of the program and is activated if the reference count decreases to zero. If an object is put in a container, e.g a dictionary/ tuple, or is given a new name the reference count increases. Likewise, when a reference to an object is deleted, reassigned or goes out of scope, the reference count decreases.

```
a = 5 #reference count of 5 increases
b = a #reference count increases
c = [b] #reference count increases
a = 4 #reference count of 4 increases ref count of 5 decreases
```

The memory allocation is a heap which accommodates objects and other data structures that are used in the program. The python memory manager controls the allocation and deallocation of the heap through the use of API functions.

[11]Python objects in memory can either be simple and contain numbers, strings, ect or be containers like dictionaries, user defined classes or lists. Variables or types don't need to be declared before being used in a program due to Python being a dynamically typed language.

The understanding of memory management in python is vastly important as python is widely used, writing efficient code frequently means writing memory-efficient python code. The significance of memory management cannot be left unnoticed with the increasing use of big data, ineffective memory management can lead to slowness on server-side and application elements. An unnecessary amount of time would be spent on testing and debugging if a memory leak were to occur. It can also cause simultaneous processing issues and create chaos on data processing. Although the python memory manager does most of the memory management, good perception into how pythons memory managers works can result in more maintainable and efficient code.

Conclusion

In the course of this report we have seen a thorough analysis of the Python programming language. However, our final conclusion is based on the phrase “Jack of all trades, master of none!” and how it relates to Python. We believe that while it is true to say Python is a “Jack of all trades”, it is unfair to say it is a “master of none!”. We have seen many examples of how Python is not a master of all things especially in its implementation and support for the OOP concept of encapsulation, it’s half-hearted support of the functional programming paradigm in comparison to languages like Haskell and how, while it succeeds in supporting abstraction, it is not a master of it, especially compared to Java. However, this only proves it is not a master of all.

In our opinion, this does not define the language. As we have said, it is a “Jack of all trades”, even if it is not a master of these topics it does indeed support them and it supports them well. But, we also believe it is untrue to say Python is a “master of none”, Python is in fact a master at supporting many things. It excels in its support for the Procedural paradigm and fully supports the Object-Oriented paradigm, we have also seen that its implementation of data types is incredibly user friendly and an aid to the developer. It even fully implements an impure Functional paradigm with help of its extensive libraries (functools). Python is ever growing in popularity and for good reason as we can see here, it is flexible and gives control to the programmer, providing them with excellent support of many tools and allowing them to choose the best use of these tools.

References

[1]A. Shukla and N. Thaker, "Python as Multi Paradigm Programming Language", https://www.researchgate.net/ , 2020. [Online]. Available: https://www.researchgate.net/publication/338632256_Python_as_Multi_Paradigm_Programming_Language . [Accessed: 03- Dec- 2021].

[2]Goldwasser, Michael H. and D. Letscher. “Using Python To Teach Object-Oriented Programming in CS1.” (2008).

[3]A. S.Gillis and S. Lewis, "What is Object-Oriented Programming (OOP)?", SearchAppArchitecture. [Online]. Available: https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP . [Accessed: 03- Dec- 2021].

[4]M. Loukides, "Multi-Paradigm Languages", O’Reilly Media, 2010. [Online]. Available: https://www.oreilly.com/radar/multi-paradigm-languages/ . [Accessed: 02- Dec- 2021].

[5]S. Bhatia, "Procedural Programming [Definition]", Hackr.io, 2021. [Online]. Available: https://hackr.io/blog/procedural-programming . [Accessed: 04- Dec- 2021].

[6]"Functional Programming - Introduction", Tutorialspoint.com. [Online]. Available: https://www.tutorialspoint.com/functional_programming/functional_programming_introduction.htm . [Accessed: 04- Dec- 2021].

[7]J. Mueller, "Embracing the Four Python Programming Styles", New Relic, 2018. [Online]. Available: https://newrelic.com/blog/nerd-life/python-programming-styles . [Accessed: 04- Dec- 2021].

[8] “Memory Management - Overview”, 2001 - 2021 Python Software Foundation. [Online]. Available: https://docs.python.org/3/c-api/memory.html . [Accessed: 04- Dec- 2021].

[9] Tusamma Sal Sabil, “Memory Management in Python”, 2020 [Online]. Available: https://medium.com/@tssovi/memory-management-in-python-d542952d5c78 . [Accessed: 04- Dec- 2021].

[10] Artem Golubin, 2020, “Garbage Collection in Python - Things you need to know”. [Online]. Available: https://rushter.com/blog/python-garbage-collector/ . [Accessed: 04- Dec- 2021].

[11] “Memory Management in Python”, 2020 . [Online] . Available: https://www.geeksforgeeks.org/memory-management-in-python/ . [Accessed: 04- Dec- 2021].

[12] AskPython. “*Understanding Abstraction in Python”*. Available at: https://www.askpython.com/python/oops/abstraction-in-python . [accessed 03/12/2021]

[13] GeeksforGeeks. “*Abstract Classes in Python”*. Available at: https://www.geeksforgeeks.org/abstract-classes-in-python/ . [accessed 03/12/2021]

[14]&[15] Aryan, R. “*Abstraction in Python Programming”.* Available at: https://pythonlobby.com/abstraction-in-python-programming/ [accessed 03/12/2021]

[16]&[17] GeeksforGeeks. *“Python Data Types”.* Available at: Python Data Types - GeeksforGeeks [accessed 04/12/2021]

[18] Real Python. “*Python Scope”.* Available at: Python Scope & the LEGB Rule: Resolving Names in Your Code – Real Python [accessed 04/12/2102]