Analytical Analysis of the Python Language

"Jack of all trades, master of none" is used to describe something that is competent in many areas but not an expert in any of them. The Python programming language can be described using this phrase.
One of the major differences between Python and other programming languages, is that Python's variables are dynamically typed while in other languages such as Java, variables are statically typed. Python's type checking is not done until runtime but in Java, variables are checked during compile time.
e.g Declaring python string variable:

```
hello = "hello"
```

e.g Declaring Java string variable:

```
String hello = "hello";
```

As can be seen from the examples above, in Python, you do not have to declare a type along with the variable name, unlike in Java which has to be done. Even though Python is therefore easier to use, this has its consequences. Python programs run slower than Java programs because of this. Its run time has to work much harder than Java's. When evaluating x+y, it has to look at the objects x and y and find out their type, which is not known at compile time. Once it does this, then the evaluation can happen. In Java, the type will already have been declared and so it can evaluate x + y immediately providing a quicker result. Not only is statically typed faster, it also prevents errors. Java can catch type errors at compile time. Python can therefore handle types well but not well enough compared to other programming languages. [1,2]

Python is an object-oriented language but not entirely, it is just a feature of it. Unlike Java, which is a purely object-oriented language. Encapsulation is one fundamental concept of object-oriented programming. It is the restriction of access to methods and variables so one can't change them elsewhere in the code. In python, we can make an attribute 'private' by [3] adding an underscore to the beginning of the attribute name. This is not actually private however, instead it is just renamed. This naming convention makes it harder to access the variable or function, but you can still access the variable or function directly. For example:

```
class Dog:

    def __init__(self):
        self.__eat()

    def bark(self):
        print('woof')

    def __eat(self):
        print('eating')

setter = Dog()
setter.bark()
#setter.__eat()  not accessible from object.
#but _setter__eat() is accessible
```

As can be seen from the example above, the private function __eat() should only be called within the class and should not be able to be called from outside the class. However, it can be called using _dog__eat(). This prevents from accessing the function accidentally but not intentionally.

In Java however, encapsulation is much more efficient. To achieve this in Java, variables or methods need to be declared as private. This will prevent other classes from accessing these variables or methods directly. Public setter and getter methods are therefore used to modify and view the values. For example:

```
public class Student {
    private String name;
    private int id;
    private String course;

    public String getName() {
       return name;
    }

    public int getId() {
       return id;
    }

    public String getCourse() {
       return course;
    }

    public void setName(String newName) {
       name = newName;
    }

    public void setId(int newId) {
       id = newId;
    }

    public void setCourse(String newCourse) {
       course = newCourse;
    }
 }
```

Following the above example, the public getter methods and public setter methods are the only way other classes can access these variables. Java therefore can handle encapsulation better than Python. Even though in Python, it does make it harder to access the variables directly, the variables are still not fully private. In Java, they are private and there is no way of accessing these variables without using the getter and setter methods provided in the class. Java is clearly a master of encapsulation, unlike Python that does it well but not as good.

In addition, in Python's object-oriented programming, class member variables are referred to using the 'self' keyword. In Java, a class refers to itself using the 'this' reference. It does not have to be written in Java as many times though, only when there are two variables with the same name. e.g.

```
public void setAccountNumber(int accountNumber) {
    this.accountNumber = accountNumber;
}
//The same can be wriiten like this
```

```
public void setAccountNumber(String newAccountNumber) {
    accountNumber = newAccountNumber;
}
//Avoiding the use of "this"
```

In Python however, ‘self’ is required if you want to create or refer to a member variable.
e.g.

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

    def get_name(self):
        return self.name
```

Using self is a bit more awkward and forgetting to put it in your code results in errors. It is therefore easier to refer to a class’s attributes in Java rather than in Python.

Similar to encapsulation, abstract classes [4,5,6,7] and abstract methods are a very important concept of object-oriented programming. An abstract class is a template for other classes. It defines a set of methods that do not have an implementation and these methods must be created within any child class built from the abstract class. These classes cannot be instantiated and must be inherited by another class. Python does not directly provide abstract classes. Instead, if you want to use abstract classes, Python comes with a library which needs to be imported. E.g

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

Importing ABC(Abstract Base Class) only works by decorating each method with ‘@abstractmethod’
E.g

```
class Animal(ABC):

    @abstractmethod
    def eat(self):
        pass
```

This decorator indicates that the following method is abstract.
In Java, however, abstract classes are much more intuitive. E.g

```
public abstract class Animal{
   public abstract void eat();
}
```

In this example, in order to declare a class as abstract, the abstract keyword is used when defining the class. The same applies for the abstract method ‘eat’-the abstract keyword is used before the return type and method name. This makes it a lot easier to use and there is 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. E.g

```
from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod
    def eat(self):
        print("eating")

class Dog(Animal):
```

```
    def eat(self):
        super().eat()
        print("dogs eat nuts")
```

Here, we can see the Animal class has an abstract method ‘eat’. Within this method, there is a body. This implementation gets overridden when the Dog class inherits this method. Java has more strict abstract methods, by not allowing any implementation in them and if provided returns with an ‘error: abstract methods cannot have a body’. Abstract methods are supposed to be templates and in Python it doesn’t hold up to the same standard. By allowing someone to provide an implantation, it defeats the original purpose of creating the abstract method. Once again, we can see how yet Python can handle abstraction well to some degree, Java is more efficient and stays true to what abstraction really is.

Inheritance [8,9] is another fundamental concept in object-oriented programming. It allows objects to derive attributes from another class, called its parent class. Once this class inherits from a parent, they are referred to as a child class. Although Python and Java both support inheritance by creating a class that inherits behaviour from a parent class, Java objects are able to inherit from interfaces too. Interfaces are like a blueprint for a class. They outline what a class must do but do not provide the body of the code stating how it does it. Interface methods are therefore abstract methods. In Java, you can only inherit from one abstract class but you can implement multiple interfaces.

```
public class MyClass implements InterfaceOne, InterfaceTwo, InterfaceThree{
}
```

This gives Java the advantage of having the ability of multiple inheritance. Java doesn’t provide multiple inheritance by default because it can lead to the diamond problem. Instead of trying to figure out that complex problem, interfaces can be used as a work around. In Python, however, when using multiple inheritance, there is a chance you could inherit the same class more than once. E.g.

```
class A:
    myAttribute = 42
class B(A):
    myAttribute = 43
class C(A):
    myAttribute = 44
class D(C,B):
    myAttribute = 45
```

“If B and C inherit from A and D inherits from B and C then it potentially gets two copies of class A”. This is the diamond problem. In Java, interfaces have the advantage of having the same default method “with the same name and signature in two different interfaces”. This allows us to implement these two interfaces and override the default methods with the correct interface name.

```
interface DemoInterface1
{
public default void display()
{
System.out.println("the display() method of DemoInterface1 invoked");
}
}
interface DemoInterface2
{
```

```
public default void display()
{
System.out.println("the display() method of DemoInterface2 invoked");
}
}
public class DemoClass implements DemoInterface1, DemoInterface2
{
public void display()
{
DemoInterface1.super.display();
DemoInterface2.super.display();
}
public static void main(String args[])
{
DemoClass obj = new DemoClass();
obj.display();
}
}
```

This example shows how both interfaces have the same method name and signature. The method is invoked by an interface name and does not create any ambiguity between methods. This achieves multiple inheritance and solves the diamond problem in a very efficient way. In Python, it is much more difficult to implement. The method resolution order (mro) has to be determined. Although the diamond problem can be solved in this way, Java's interfaces are a work around that allows you to not have to think about the diamond problem at all.

Although Python is capable of functional programming, it only supports some features of a functional programming language. On the other hand, Haskell is a pure functional programming language. Haskell has support for immutability [10] which Python does not cater for. Immutability is the idea that once an expression has been assigned to a name, its value can't change. Mutability in Python can be seen below:

```
def reverse_a():
    x = [4,5,6]
    x.reverse()
    #print x
    #x = [6, 5, 4]
    y = x.reverse()
    #print x
    #x = [6,5,4]
    #print y
    #= (No output)
```

When we call reverse on "x", the value of "x" changes. We can see this by printing x and seeing how the value changed from [4,5,6] to [6,5,4]. When we try to assign y to equal x.reverse(), we get no output. This occurs because the reverse function doesn't have a return value. In Haskell however:

```
let x = [4,5,6]
reverse x
--x = [6,5,4]
```

--call x
--x = [4,5,6]

The value of x did not change. Reverse worked but it did not have an effect on the original variable. To access the reverse list, we would have to use 'let' and assign it to a new name. Looking at the type declared in the Haskell reverse function
Reverse :: [x] -> [x]
We can see that the function takes a list of type x and outputs a list of the same type. This shows that it is a pure function as the input list can't be changed. A new list is created for the output. This contains the same elements as the input but in reverse order. In Python, each function could change the original input, so we do not know what is passed to the next function. However, in Haskell, we know the exact same input will be passed to all the functions used. This guaranteed immutability prevents bugs, is easier to refactor and allows for persistent data structures. These would be very hard to achieve in Python as it does not support immutability.

Tail call optimization [11] is another essential feature of functional programming that Python does not support. Without this, deep recursion is not possible. A function call is recursive when it's done inside the function being called. It's a function calling itself. Tail recursion is where the return value of the function is calculated as a call to itself and nothing else. Python can define recursive functions e.g.

```
def recursive_factorial(n):
   if n == 1:
       return n
   else:
       return n*recursive_factorial(n-1)
```

and tail recursive functions e.g.[12]

```
def factorial(n, acum=1):
    if n == 0:
        return acum
    else:
        return factorial(n-1, acum*n)
```

This tail recursive function works perfectly fine for small numbers as there is a small amount of recursive calls. Unfortunately, without tail call optimization, Python will get a stack overflow error. This occurs because Python has a limited amount of recursive calls that it can perform. Recursive calls takes up a lot of memory and resources as "each frame in the call stack must be persisted until the call is complete". This is why tail call optimization is absolutely necessary. When I attempted to get the factorial(1000) using the tail recursive function from above, I got the following error:
'RecursionError: maximum recursion depth exceeded in comparison'. Python therefore can't handle recursion well and a reason for this is because it was not built with recursion in mind, it was built around iteration.
Haskell supports tail call optimization and so tail recursive functions will run without a stack overflow error. When a tail recursive function is about to return, its environment can be discarded and the recursive call can be executed without a new stack frame being created. e.g of tail recursive function:

```
factorial :: (Eq x, Num x) => x -> x
factorial 0 = 1
```

factorial a = a * factorial (a - 1)

This function uses tail call optimization and this allows tail recursive functions to indefinitely recur unlike in Python which would run into a stack overflow. I ran this function 'factorial 1000' which in python couldn't evaluate but in Haskell it ran perfectly. As recursion is such a critical feature for functional programming, and tail recursive functions are needed, Python can't quite hold up to other functional programming languages such as Haskell. Python is limited in what it can do recursively and this is why it's not a master in functional programming.

One of the most defining aspects of pure functional languages is referential transparency [13]. An expression is referentially transparent if given a function and an input value, the function will always result in the same output. E.g in Python:

```
def plus_one(x):
    return x + 1
```

This function is referentially transparent because, given an input, you could replace that with a value instead of calling the function. Instead of calling plus_one(10), we could just replace that with 11 as 10 + 1 = 11.
The Python language does not enforce this behaviour though. It also allows for non-referentially transparent functions e.g

```
total = 0
def add(x):
    global total
    total += 1
    return x + total
```

If we were to call add(x) + add(x) this would not produce the same value as calling 2 * add(x). This is because the function add(x) uses and modifies a global variable (total). Therefore, when we invoke this function multiple times, the result depends on the changing state and the argument supplied.
Haskell, being a pure functional language, enforces this behaviour. E.g

"f :: Int -> Int"

For any integer x, it is always true that 2 * (f x) == (f x) + (f x). Referential transparency is a very powerful tool and makes programs easier to reason about. Referential transparency should eliminate side effects from occurring. Unfortunately for Python, because you can define non-referential functions, side effects can occur. This happens because as we saw the function was being modified from outside by a global variable. This breaks referential transparency and you lose the ability to use that function in any context. As Haskell supports this feature, it allows any function to be substituted with a different implementation depending on the context. Referentially transparent functions also support immutability and lazy evaluation in Haskell. It is key to pure functional programming languages and yet again Python can perform referential transparency, but it doesn't hold up compared to Haskell.

Parameter passing [14] in Python is also limited to one form – pass-by-value. C++ supports both pass-by-reference and pass-by-value. Pass-by-value is when the value of a function parameter is copied into another location. When this value is accessed within the function, only the copy is accessed or modified, the original is left unchanged. E.g

```
age = 21
def birth_year(age):
```

```
    birth_year = this_year - age
    return birth_year
```

As can be seen in the above example, age is passed into the function birth_year as an argument. This argument 'age' is copied and so the original variable that we defined will not be affected by any modifications that could occur in the function. The only way to change the original variable is by using the return value of the function.
While C++ also supports pass-by-value, unlike Python, it also supports pass-by-reference. E.g

```
function increaseAgeByRef(int &age) {
    *age = *age + 1;
}

int age = 21;
increaseAgeByRef(age);
```

Following the example, pass-by-reference occurs when the memory address of a variable is passed to the function. The variable age in this function is pointing to the memory address of the original 'age' variable and so any changes made to the variable in the function results in the original variable changing. The '&' in '&age' refers to a reference parameter so the function is aware we are passing a reference. This form of parameter passing has many advantages over pass-by-value. References allows a function to change the value of the argument which can be very useful. In the above case, age, which was originally set to 21, would change to 22 after one call of the function. There was no return value for this function as the reference changed the age variable. If one doesn't want this behaviour, const references can be used. These guarantee that the original variable can't be modified by the function. When Python uses pass-by-value, a copy has to be made. In C++ however, using references eliminates the copying of the argument making this method of parameter passing very fast even when used with large structs or classes. References can be used to return multiple values from a function. This can't be done using pass-by-value which only allows one return value. In terms of parameter passing, C++ is therefore more proficient than Python.

As you can see, Python is competent in many areas ranging from data types, encapsulation, class-member variables, abstract types, inheritance, functional programming, and parameter passing. However, even though it is capable of these important aspects of programming, it does not do all these as great as other programming languages. It is therefore not a master but a jack of all trades.

Word count (Excluding references and examples) = 2,666

Reference list:

[1] Python vs Java. [Online]. Available: https://www.snaplogic.com/glossary/python-vs-java [Accessed: December 14th 2020].

[2] Comparing Python to Other Languages. [Online]. Available: https://www.python.org/doc/essays/comparisons/. [Accessed: December 14th 2020]

[3] Encapsulation. [Online]. Available: https://pythonspot.com/encapsulation/. [Accessed: December 15th 2020].

[4] Python Tutorial: 'The ABC' of Abstract Classes. [Online] Available: https://www.python-course.eu/python3_abstract_classes.php. [Accessed: December 15th 2020].

[5]GeeksForGeeks. Abstract classes in Python.[Online] Available: https://www.geeksforgeeks.org/abstract-classes-in-python/. [Accessed: December 15th 2020].

[6] The Medium. Abstract classes in Python. [Online] Available: https://medium.com/techtofreedom/abstract-classes-in-python-f49cf4efdb3d. [Accessed: December 15th 2020].

[7] Why Abstract class is important in Java. [Online] Available: https://www.java67.com/2014/06/why-abstract-class-is-important-in-java.html [Accessed: December 15th 2020].

[8] Programmer's Python -Multiple Inheritance. [Online] Available: https://www.i-programmer.info/programming/python/12217-programmers-python-mutliple-inheritance.html?start=1#:~:text=In%20Python%20as%20all%20classes,the%20simplest%20of%20multiple%20inheritance. [Accessed: December 16th 2020].

[9] What is Diamond Problem in Java. [Online] Available: https://www.javatpoint.com/what-is-diamond-problem-in-java. [Accessed: December 16th 2020].

[10] Immutability is Awesome. [Online] Available: https://mmhaskell.com/blog/2017/1/9/immutability-is-awesome. [Accessed: December 16th 2020].

[11] Tail recursion in Python. [Online] Available: https://chrispenner.ca/posts/python-tail-recursion#:~:text=Some%20programming%20languages%20are%20tail,only%20a%20call%20to%20itself. [Accessed: December 17th 2020].

[12] Python Program to Find Factorial of Number using Recursion. [Online] Available: https://www.programiz.com/python-programming/examples/factorial-recursion. [Accessed: December 17th 2020].

[13] What is referential transparency. [Online] Available: https://softwareengineering.stackexchange.com/questions/254304/what-is-referential-transparency. [Accessed: December 18th 2020].

[14] Passing by Value vs. by Reference Visual Explanation. [Online] Available: https://blog.penjee.com/passing-by-value-vs-by-reference-java-graphical/#:~:text=The%20terms%20%E2%80%9Cpass%20by%20value,where%20the%20value%20is%20stored. [Accessed: December 19th 2020].