THEORY QUESTIONS ASSIGNMENT

Software Stream

Maximum score: 100

KEY NOTES

- This assignment to be completed at student's own pace and submitted before given deadline.
- There are 10 questions in total and each question is marked on a scale 1 to 10. The maximum possible grade for this assignment is 100 points.
- Students are welcome to use any online or written resources to answer these questions.
- The answers need to be explained clearly and illustrated with relevant examples where necessary. Your examples can include code snippets, diagrams or any other evidence-based representation of your answer.

Theory questions	**10 point each**

1. How does Object Oriented Programming differ from Process Oriented Programming?

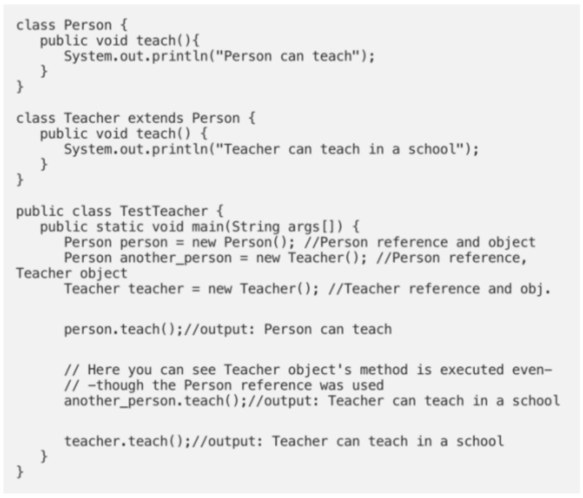
The difference between Object Oriented Programming (OOP) and Process Oriented Programming, also referred to as procedural programming, is that first and foremost, OOP relies on the principle of representation of objects through codes through their properties and their methods, it is mostly used by language such as Java, C++, C#, Python, among others (GeeksforGeeks, 2019). On the other hand, procedural programming refers to the idea of basing the structure of the code by the routines and functions on how they call each other to achieve specific tasks, i.e., it is "based upon the concept of calling procedure" (GeeksforGeeks, 2019).

So, some of the key differences between then both is that OOP breaks the code into objects, whereas procedural programming uses records and focuses on units of functions or processes. OOP provides a way to handle access to variables or methods by making them private allowing for secure code however procedural programming does not have this capability. Finally, one key difference between the two approaches is that procedural adopts a top-down approach by including instructions and routines for the computer to accomplish, whereas OOP follows the creation of objects and uses key pillars of OOP, inheritance, abstraction, polymorphism, and encapsulation to adopt a bottom-up approach (Lili Ouaknin Felsen, 2017).

2. What's polymorphism in OOP?

Polymorphism in OOP is defined by its make-up as *poly* meaning 'many' and *morph* meaning 'form' – which together mean describing something that can take many forms (Sumo Logic, 2021). It is a key feature of OOP whereby it relates to how different objects are able to be used by a single function, for example the len() function is a function that is applicable to a key number of specific objects or data types, such as lists or strings or dictionaries.

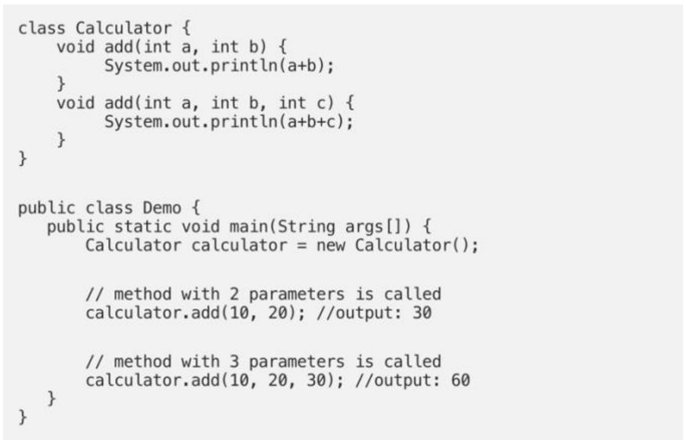
There are different types of polymorphism, the main two however are **dynamic polymorphism** and **static polymorphism** (Ediriweera, 2017). Dynamic polymorphism, or runt-time polymorphism, utilises method overriding, whereby methods of the same name within a class hierarchy, usually a parent and child hierarchy. The method however that will run will be the object's and not the method of the reference class, as seen in the example below. Essentially "Polymorphism ensures that the proper method will be executed based on the calling object's type" (Nerd.vision, 2021).



```
class Person {
   public void teach(){
      System.out.println("Person can teach");
   }
}

class Teacher extends Person {
   public void teach() {
      System.out.println("Teacher can teach in a school");
   }
}

public class TestTeacher {
   public static void main(String args[]) {
      Person person = new Person(); //Person reference and object
      Person another_person = new Teacher(); //Person reference,
Teacher object
      Teacher teacher = new Teacher(); //Teacher reference and obj.

      person.teach();//output: Person can teach

      // Here you can see Teacher object's method is executed even-
      // -though the Person reference was used
      another_person.teach();//output: Teacher can teach in a school

      teacher.teach();//output: Teacher can teach in a school
   }
}
```

Figure 1 Example of dynamic polymorphism (Ediriweera, 2017)

Static polymorphism on the other hand, is what can be described as essentially method overloading. It occurs when more than one method has the same name but different functionalities, and so the compiler will decide which method to call depending on which method is overriding the other, therefore static polymorphism is also at times referred to as compile-time polymorphism, or early-binding, or even just method overloading.



```
class Calculator {
    void add(int a, int b) {
         System.out.println(a+b);
    }
    void add(int a, int b, int c) {
         System.out.println(a+b+c);
    }
}

public class Demo {
   public static void main(String args[]) {
       Calculator calculator = new Calculator();

       // method with 2 parameters is called
       calculator.add(10, 20); //output: 30

       // method with 3 parameters is called
       calculator.add(10, 20, 30); //output: 60
   }
}
```

Figure 2 Example of static polymorphism (Ediriweera, 2017)

It is important to be aware of polymorphism and the different types because one of the downsides to it is that it can be used to create cyber-attacks through creating malicious code or scripts e.g., creating malware. Researchers have also found that 97% of malware attacks and infections are polymorphic (Milena Dimitrova, 2016), i.e., they rely on polymorphism. Therefore, organisations now need to become better equipped to recognise polymorphic malware and threats, not just traditional threats, and malware.

3. What's inheritance in OOP?

Inheritance is when a class, the child class, can inherit the methods and properties of another class, which is referred to as the parent class. The child class can access all the methods and properties of the parent class however it can also override the functionality of these methods and properties which will take precedence when being used in the program, e.g., calling a method that is named the same in both classes, the child class function will run. Inheritance is an incredibly powerful concept as it allows for abstract objects to hold methods and properties shared by many other objects that have been abstracted to one parent class and can be used to quickly and easily create newer classes that have access to all its functionality and properties. It also means that we can build a hierarchy of classes that can have one parent, share parents, or even have multiple parents where applicable.

4. If you had to make a program that could vote for the top three funniest people in the office, how would you do that? How would you make it possible to vote on those people?

I would first analyse the specifications required for this program and highlight what may become potential classes.

If you had to make a program that could vote for the top three funniest people in the office, how would you do that? How would you make it possible to vote on those people?

From this quick analysis of the questions, I have identified three key components. (1) the people, a potential class, whereby the person or the employee class will represent every person working in the office. (2) vote, an abstract class that will hold the key methods that allows a person to vote for a person between a selection of people, this will be applied to the people class as its parent class, giving people that functionality, so that if I would ever need to modify that voting functionality, e.g., if I want the employee to vote for 5 people - I can modify it all from the key abstract class. (3) the office class will hold all the people in the office and be used to manage the employees and collect and present the results. I would also need some python files to handle calls to the database to collect the names and employee_ID's in order to use for the voting system. After identifying my 3 classes, I would brainstorm the necessary properties and methods they would have, e.g.:

1. Employee (Vote)
Properties
- Name - Employee_ID - Department
Methods
- Vote (abstract method) *This vote method will envoke the super.vote(employee_ID)*

2. Vote
Properties
- Top_three_votes = [*employee_ID, employee_ID, employee_ID*]
Methods
- Vote (employee_ID) -ABSTRACT METHOD - Get_votes_from_db - Add_one_vote - Calculate_top_three_votes - Get_top_three_votes() – GETTER MTHOD

3. Office (Vote)
Properties
- Employees = [employee_ID…] - Top_three_employees = [] - Department
Methods

- Get_top_three_votes(abstract method)

This vote method will envoke the super.Get_top_three_votes() from Vote class

The key functionality here lies inside the Vote class. The Vote function will take in the employee ID, allow them to vote for three separate employee by inserting their employee codes, and then that vote entry is stored to the database as their ID, and the ID's of the people they voted for in a list {Employee_ID: [employee_ID, employee_ID, employee_ID]}, this is so that a record of all votes made is kept for future reference. There will be a specific 'scores' table where every employee ID is listed as one column – the primary key, and a score is associated in the next, all scores start at 0, every time the Vote method is called in the Employee class, the three votes, i.e. the employee_IDs list is passed into the Add_one_vote method which will send a specific query to update each 'vote' column by 1 for the associated primary key column in that table, which is the employee_ID. The Calculate top three votes is called when someone wants to access the current top three, this is done through a database query that will return the top three highest scoring employee_ID's from the scores table by ordering it by the score value and limiting to the top three entries. The returned result will be stored into the Vote class property as top_three_votes and will be returned to the Office class which calls this method to the see the latest top 3 funniest people in the office.

5. What's the software development cycle?

The software development cycle is a framework that comprises of the stages of development of a piece of software typically used in software development. It is these common stages that are adopted into different methodologies such as waterfall or agile. The stages of the software development life cycle or the SDLC are (1) the planning stage, (2) the analysis stage, (3) the design stage, (4) the implementation stage, (5) the testing and integration stage, and finally (6) the maintenance stage until the next cycle starts (phoenixNAP Blog, 2019). Sometimes an extra stage is added to identify the deployment stage as seen in figure 3. In the planning stage, we map out the scope of the project, plan for all the requirements of the program, functional and non-functional, and prepare timelines and estimations for the project. The analysis stage is where the project goals are created, and the inner workings of the system

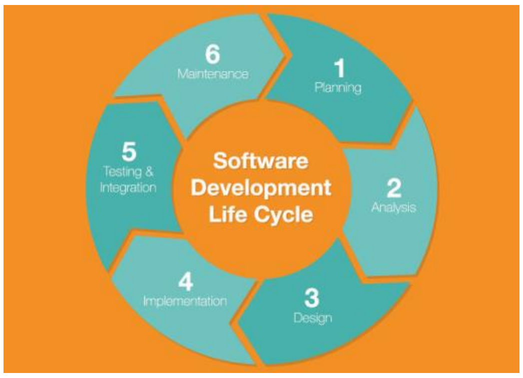


Figure 3 Lifecycle Development Cycle (Harvey, 2020)

are now being decided. The next stage is the design stage, this is where the architecture is designed, the users' stories are created in this stage that will inform the tasks for the backlog, this is also where the prototypes are created for the software itself. Next is the development and implementation stage where the project is coded during the sprints in what may be iterations depending on the engineering approach adopted by the team. The stage after that includes the testing and integration stage where the system is tested in many ways such as unit testing or regression testing to ensure that the system is durable and built to the requirements and specifications of the product owner. It is then deployed in what could be a stage of the SDLC, as seen in figure 4, as it is not always defined as a stage on its own. Finally, the last universal SDLC stage is the maintenance stage whereby the system will be continuously maintained and updated also it is where any errors and bugs that arise during the time where users are now using the software arise are now fixed and resolved. These main stages of software development are used to create many different approaches to software engineering such as the agile school of thought or the waterfall methodology.

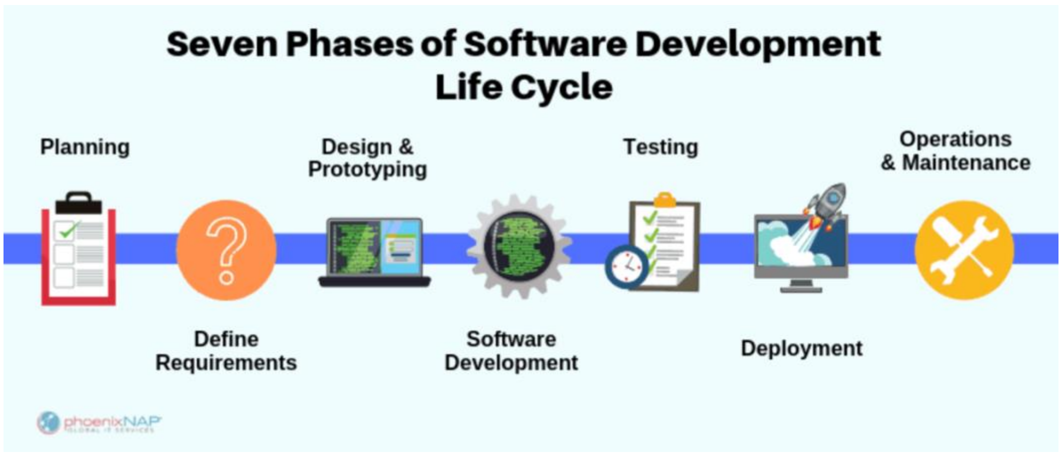


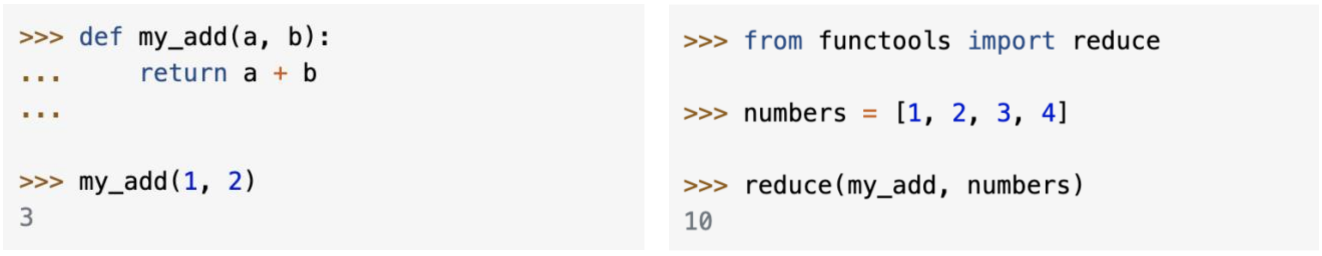
Figure 4 Seven phases of software development (phoenixNAP Blog, 2019)

6. What's the difference between agile and waterfall?

The key difference between agile and waterfall first and foremost is their approach to development style, where agile is free spirited and unstructured, waterfall is discipled and rigid and values structure. The agile manifesto was created to allow developers fee will and a lot of power to work on projects in an agile manner, where the projects are built in iterations and each release refines the project further. It is good for app productions or getting an MVP quickly, so it is an approach adopted by start-ups frequently, it allows for quick turnaround of an MVP compared to a fully realised and polished product. On the other hand, the waterfall discipline is structured in the stages and the iterations it goes through from planning all the way down to the final release. The idea is that once a stage has been completed then the team will move to the next stage, and not go back to the first stage, forward and down to the end not backward – just like a waterfall. This type of approach requires detailed plans, thought out precautions and any error handling coverage to be completed as much as possible before even starting to code the project. This type of approach is very well suited to high-risk projects, especially where health and safety is involved, so it can be seen used in the military or in aeroplanes.

7. What is a reduced function used for?

The reduce function is based upon a mathematical technique also referred to as **folding** or **reduction** (Real Python, 2020). It is a popular method to use when programming in a functional style of programming. The function itself can be called with any iterable object, not necessarily just a list. It works to first apply a function that is passed into it as its first argument on the first two items within the iterable object and to receive the result of that, the next step is to use that result from the first step alongside the second item within the iterable object back into the callable function and receive another result, and finally this whole process is repeated until one final result is left behind which is the result of the reduce function overall. Essentially the reduce function will apply a callable function cumulatively to every item within the iterable object passed into it and return a single value. The reduce function can also accept another argument alongside the callable function and the iterable object, which is an initialiser, which would be used as the initial item to be passed into the callable function with the first item of the iterable object, instead of just starting with the first two items within the iterable object. As function is written in C, the language python was also created with, it is much faster than creating for loops in python to do the exact same task (Real Python, 2020).



```
>>> def my_add(a, b):
...     return a + b
...

>>> my_add(1, 2)
3
```

```
>>> from functools import reduce

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

>>> reduce(my_add, numbers)
10
```

Figure 5 Summing the numbers in a list using reduce (Real Python, 2020)

Some of the common use cases for the reduce function include summing the values of a list using just the reduce function and a predefined add function that will add one number to another, as seen in figure 5.

Another use case includes finding the minimum and maximum values from within a list, checking if all values within a list are True, or even checking if any values in a list are True (Real Python, 2020). What's great about using the reduce function is that it won't mutate the objects passed through it and will always return a single value (Babak, 2019).

8. How does merge sort work

Merge sort is an example of a divide and conquer algorithm and is "one of the most prominent *divide-and-conquer* sorting algorithms in the modern era" (Edpresso Team, 2019). It can sort values in any traversable data structure, such as a list for example. It utilises the power of recursion in order to continuously break down the structure in halves until it reaches lists of ones, i.e. each item is within its own list. Once we have each item in it's own list, we can compare the heads of the lists to each other and the smallest item will be placed first when combined together again. This process of checking each pair continuous on the same level until we have lists of two items each. The process of comparing heads of the lists starts again until weare left with one complete and sorted list. The action of comparing the lists then combining them in the order of smaller to larger is the **merge**.

The code implementation is shown after the illustration:

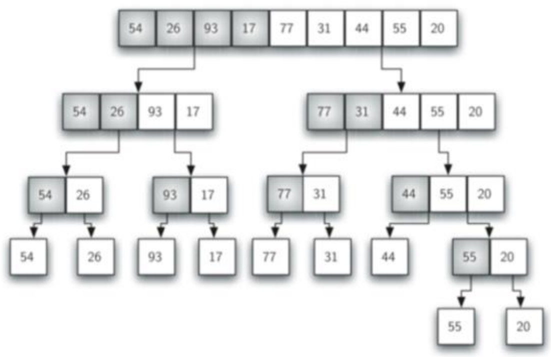


Figure 10: Splitting the List in a Merge Sort

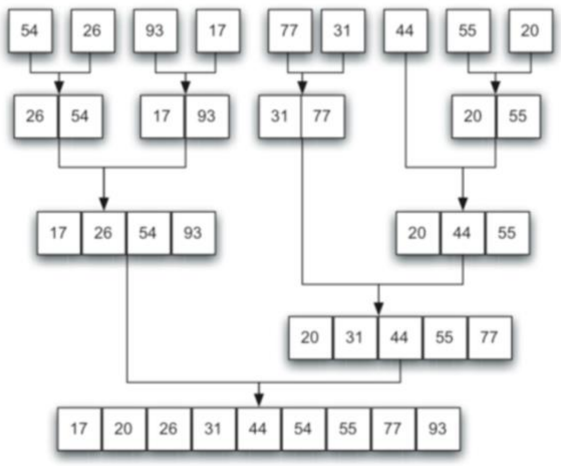


Figure 11: Lists as They Are Merged Together

Figure 6 Merge sort illustrated (Runestone.academy, 2014)

```
def mergeSort(alist):

   if len(alist)>1:
       mid = len(alist)//2
       lefthalf = alist[:mid]
       righthalf = alist[mid:]

       #recursion
       mergeSort(lefthalf)
       mergeSort(righthalf)

       i=0
       j=0
       k=0

       while i < len(lefthalf) and j < len(righthalf):
           if lefthalf[i] < righthalf[j]:
               alist[k]=lefthalf[i]
               i=i+1
           else:
               alist[k]=righthalf[j]
               j=j+1
           k=k+1

       while i < len(lefthalf):
           alist[k]=lefthalf[i]
           i=i+1
           k=k+1

       while j < len(righthalf):
           alist[k]=righthalf[j]
           j=j+1
           k=k+1

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

This algorithm has a worst case and average case complexity of $O(n\log n)$ algorithm. The benefits of using merge sort are that it is fast and efficient, especially compared to quick sort in terms of working on larger datasets (GeeksforGeeks, 2018). However, some downsides to this algorithm is that it ‘requires extra space to hold the two halves as they are extracted with the slicing operations’ (Runestone.academy, 2014), so while it worst case factor isn’t that bad, this algorithm will require much large space for larger datasets, whereas in comparison, quick sort sorts in place and doesn’t suffer from this issue. Merge sort has also been described as a stable algorithm, and this is because ‘two elements with equal value appear in the same order in sorted output as they were in the input unsorted array’ (GeeksforGeeks, 2018). While it has its benefits, it is much slower compared to other sort algorithms when it comes to smaller tasks.

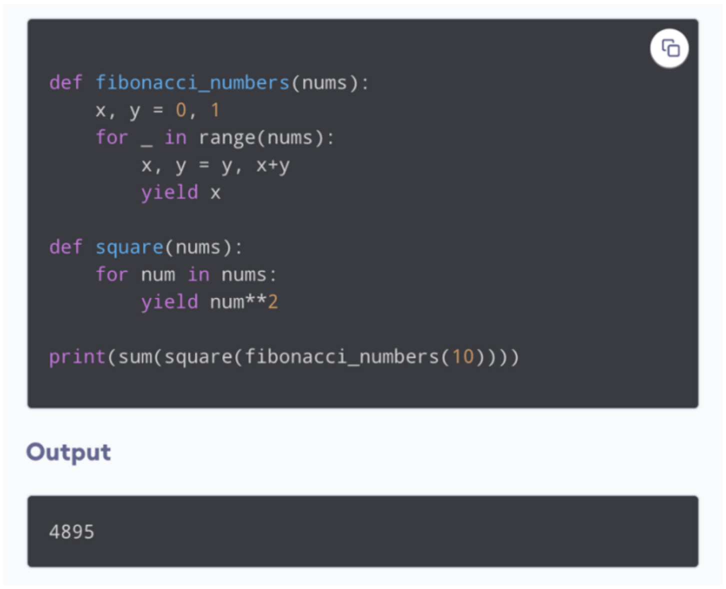
9. Generators - Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop. What is the use case?

A generator is a function that behaves and acts like an iterator but the key differences between the two is that the iterator is an object that is implemented through a class, whereas a generator is created using a function, local variables aren’t used in iterators but they can be found in generators before the yield keyword, also where iterators are used to iterate through items using the iter() function – generators are mostly used in loops to generate an iterator through returning the values in the loop without it affecting the iteration of the loop, and finally but most importantly, not iterators are not generators, but a generator is an iterator (GeeksforGeeks, 2021).

By stating the difference, we can now discuss how generators behave like iterators? Generators provide a way to create custom iterators through just a written function. Essentially, a ‘generator is a function that returns an object (iterator) which we can iterate over (one value at a time)’ (Programiz.com, 2021). Generators are created by writing a function and adding a yield keyword instead of the return keyword. The function below is an example of a generator to create a sequence of power of two:

```
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Some of the key benefits to using generators over iterators, is that it is much easier to implement, iterators need to be created through a class, whereas the generator can do the same job with much fewer lines of code with one function alone. As the generator only produces one item at a time, it is much more memory friendluy compared to an iterator which will create the entire sequence before returning the result. Following from that, it also means that generators are great at representing infinite streams of data, as it is something that cannot be stored in memory, for it is infinite, but because generators can produce items one at a time, then it can produce them infinitely. Finally, a powerful benefit of using generators is that they can be used to pipeline a series of operations, for example finding out ‘the sum of squares of numbers in the Fibonacci series’, this can be found by creating two separate generator functions, one to create the Fibonacci series and another that will square a num, then place pipe those into the sum() function to receive a final number as shown in the picture below:


```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci_numbers(10))))
```

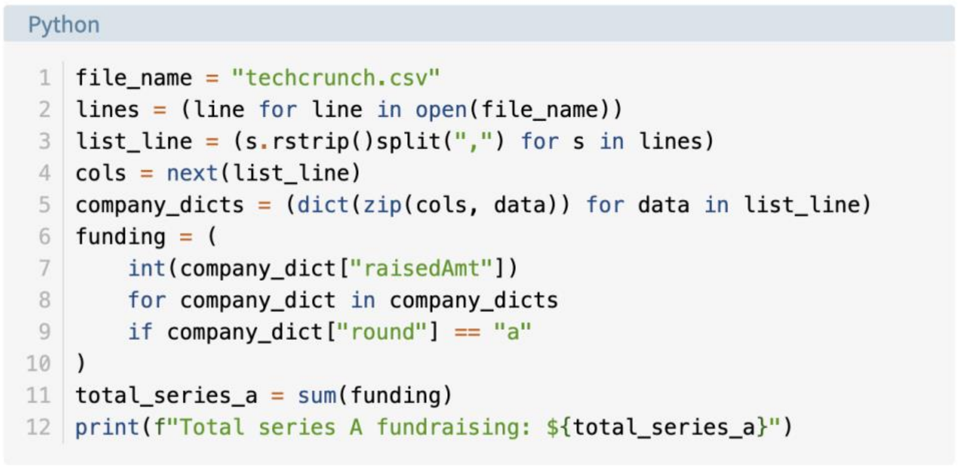
Output

```
4895
```

Figure 7 Pipelining generators (Programiz.com, 2021)

By discussing the benefits of generators, we can now start to understand the use cases for this type of code. One main use case includes reading in large text files (AskPython, 2019). A test conducted by AskPython (2019) found that a 263MB file used 3.65 GB of memory and took 273.56s to run as a normal return function, whereas the generator function was able to read that same file with 5.55MB of memory in 292.99s. The memory used by the return statement only increased the more the file size increased, whereas it remained generally the same with the generator function (AskPython, 2019).. This shows that it is memory efficient in reading large files compared to a normal return function, while also being easy and fast to implement.

Another real-world use case for generators includes data pipelines and processing large datasets. Real Python (2019) described a 12 line script that will read each line from a CSV file containing information regarding companies and their funding in the USA using a generator expression, then the script will split lines into values and place them into a list, to then create them into dictionaries, and from there it will filter amounts raised keeping only the funding amounts and then will finally sum the final funding’s necessary to then return the total series A fundraising (Real Python, 2019). The code described is shown below:



```python
file_name = "techcrunch.csv"
lines = (line for line in open(file_name))
list_line = (s.rstrip()split(",") for s in lines)
cols = next(list_line)
company_dicts = (dict(zip(cols, data)) for data in list_line)
funding = (
    int(company_dict["raisedAmt"])
    for company_dict in company_dicts
    if company_dict["round"] == "a"
)
total_series_a = sum(funding)
print(f"Total series A fundraising: ${total_series_a}")
```

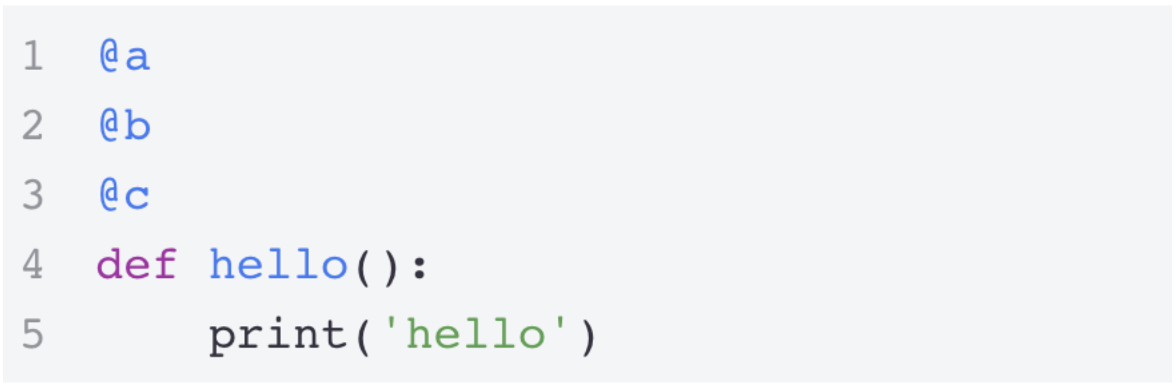
Figure 8 Example of generator expressions and data pipeline (realPython, 2019)

While a csv or pandas library would be better suited for these operations, it represents what can be done with generators and a use case for using generator expressions.

10. Decorators - A page for useful (or potentially abusive?) decorator ideas. What is the return type of the decorator?

Essentially a decorator is ‘simply a function that is passed a function, and returns an object’ (Humrich, 2018). A decorator is created by wrapping the decorator function around the function that will be executed, as the nested function. This nested function will be the callable function when we use the decorator over other functions.

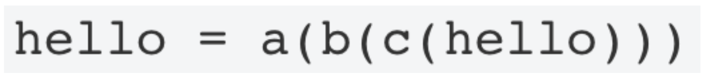
The order in which decorators are placed over a function is also very important. As each decorator wraps around the other functions and therefore this piece of code:



```python
@a
@b
@c
def hello():
    print('hello')
```

Figure 9 (Humrich, 2018)

Can also be understood as this:



```
hello = a(b(c(hello)))
```

Figure 10 (Humrich, 2018)

Decorators are used as a design patterns to ‘allow a user to add new functionality to an existing object without altering its structure’ (Tutorialspoint.com, 2021). Some of the biggest benefits of creating and building custom decorators is that it can change the behaviour of how a function works, without changing the actual lines within those functions, while also greatly improving the readability of your own code (Humrich, 2018). Some examples of the use case for decorators include performing extra steps for setup or cleanup around a method, pre-process the method arguments, or even post-process the return values (Python-patterns.guide, 2018).

There are also many levels to the types of decorators that can be created, such as the mainly mentioned method decorators, we can also have class decorators, which is a decorators declared just before a class and is used to ‘observe, modify, or replace a class definition’; we can also have an accessor decorator, which is declared just before an accessor declaration, and can be used to get and set that property as seen in JavaScript, and we can also have parameter decorators which is used in typescript often (typescriptlang, 2021). So while there are different types of decorators, the main two that can be found in python especially are the function/method decorators and the class decorators.

One argument used against decorators, where some may claim that people may abuse its power, is in the fact that python should be readable, and decorators, while re-use code, take-away from its readability. Another point to note is that in using a decorator on a function, then the ‘function itself is effectively lost from the namespace and there’s no (simple) way to call it in its original un-decorated form’ (Sagalaev, 2012). An issue maybe raised with this in that we can’t anticipate that that piece of code will never need to be called again in an undecorated way. One recommendation is to provide a decorated version and an undecorated version of the came code and allow the user to decide how to call it (Sagalaev, 2012).

In essence to answer the question of what the return type of a decorator is, then in the case of a function decorator, then the return type of a modified function, and in a class decorator then the return type is a modified class (Python-course.eu, 2011).

Reference List

AskPython. (2019). *Python yield - Generator Function Real Life Examples - AskPython*. [online] Available at: https://www.askpython.com/python/python-yield-examples [Accessed 11 Aug. 2021].

Babak (2019). *Why you should use reduce instead of loops -- Part I*. [online] DEV Community. Available at: https://dev.to/babak/why-you-should-use-reduce-instead-of-loops----part-i-5dfa [Accessed 5 Aug. 2021].

Ediriweera, S. (2017). *Polymorphism explained simply! - Shanika Ediriweera - Medium*. [online] Medium. Available at: https://medium.com/@shanikae/polymorphism-explained-simply-7294c8deeef7 [Accessed 5 Aug. 2021].

Edpresso Team (2019). *Merge sort in Python*. [online] Educative: Interactive Courses for Software Developers. Available at: https://www.educative.io/edpresso/merge-sort-in-python [Accessed 11 Aug. 2021].

GeeksforGeeks. (2021). *Difference Between Iterator VS Generator - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/difference-between-iterator-vs-generator/ [Accessed 11 Aug. 2021].

GeeksforGeeks. (2019). *Differences between Procedural and Object Oriented Programming - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/ [Accessed 5 Aug. 2021].

GeeksforGeeks. (2018). *Quick Sort vs Merge Sort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ [Accessed 11 Aug. 2021].

Harvey, S. (2020). *Best Practices for a Secure Software Development Life Cycle (SDLC)*. [online] KirkpatrickPrice Home. Available at: https://kirkpatrickprice.com/blog/what-is-a-secure-software-development-life-cycle/ [Accessed 4 Aug. 2021].

Humrich, Nick. (2018). *Decorators in Python: What you need to know*. [online] Available at: https://timber.io/blog/decorators-in-python/ [Accessed 11 Aug. 2021].

Lili Ouaknin Felsen (2017). *Functional vs Object-Oriented vs Procedural Programming*. [online] Medium. Available at: https://medium.com/@LiliOuakninFelsen/functional-vs-object-oriented-vs-procedural-programming-a3d4585557f3 [Accessed 5 Aug. 2021].

Milena Dimitrova (2016). *97% of Malware Infections Are Polymorphic, Researchers Say - How to, Technology and PC Security Forum | SensorsTechForum.com*. [online] How to, Technology and PC Security Forum | SensorsTechForum.com. Available at: https://sensorstechforum.com/97-of-malware-infections-are-polymorphic-researchers-say/ [Accessed 5 Aug. 2021].

Nerd.vision. (2021). *Polymorphism, Encapsulation, Data Abstraction and Inheritance in Object-Oriented Programming | nerd.vision*. [online] Available at: https://www.nerd.vision/post/polymorphism-encapsulation-data-abstraction-and-inheritance-in-object-oriented-programming [Accessed 5 Aug. 2021].

phoenixNAP Blog. (2019). *What is SDLC? How the Software Development Life Cycle Works*. [online] Available at: https://phoenixnap.com/blog/software-development-life-cycle [Accessed 4 Aug. 2021].

Programiz.com. (2021). *Python yield, Generators and Generator Expressions*. [online] Available at: https://www.programiz.com/python-programming/generator [Accessed 11 Aug. 2021].

Python-course.eu. (2011). *Python Advanced: Easy Introduction into Decorators and Decoration*. [online] Available at: https://www.python-course.eu/python3_decorators.php [Accessed 12 Aug. 2021].

Python-patterns.guide. (2018). *The Decorator Pattern*. [online] Available at: https://python-patterns.guide/gang-of-four/decorator-pattern/ [Accessed 12 Aug. 2021].

Real Python (2019). *How to Use Generators and yield in Python*. [online] Realpython.com. Available at: https://realpython.com/introduction-to-python-generators/#creating-data-pipelines-with-generators [Accessed 11 Aug. 2021].

Real Python (2020). *Python's reduce(): From Functional to Pythonic Style*. [online] Realpython.com. Available at: https://realpython.com/python-reduce-function/ [Accessed 5 Aug. 2021].

Runestone.academy. (2014). *6.11. The Merge Sort — Problem Solving with Algorithms and Data Structures*. [online] Available at: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html [Accessed 11 Aug. 2021].

Sagalaev, I. (2012). *When to use decorators in Python*. [online] Softwaremaniacs.org. Available at: https://softwaremaniacs.org/blog/2012/07/09/when-to-use-decorators/ [Accessed 11 Aug. 2021].

Sumo Logic. (2021). *What is Polymorphism in Computer Science? | Sumo Logic*. [online] Available at: https://www.sumologic.com/glossary/polymorphism/ [Accessed 5 Aug. 2021].

Tutorialspoint.com. (2021). *Python Design Patterns - Decorator*. [online] Available at: https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_decorator.htm [Accessed 12 Aug. 2021].

typescriptlang. (2021). *Documentation - Decorators*. [online] Available at: https://www.typescriptlang.org/docs/handbook/decorators.html [Accessed 12 Aug. 2021].