

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? [done]
2. What's polymorphism in OOP? [done]
3. What's inheritance in OOP? [done]
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? [done]
5. What's the software development cycle? [done]
6. What's the difference between agile and waterfall? [done]
7. What is a reduced function used for? [part-done]
8. How does merge sort work [23]
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? [23]
10. Decorators - A page for useful (or potentially abusive?) decorator ideas. What is the return type of the decorator? [24-28]

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

Process Oriented Programming is a programming model whereby the program is divided based on function. That means, the program is broken into step-by-step processes to be executed by the computer. Process Oriented Programming takes a top-down approach, which means the program follows a step-by-step approach to address issues individually.

Object Oriented Programming is a programming model whereby the program is divided based on the concept of an "object". These objects in the program are identified and given behaviours (methods) and characteristics (values) in a way that can be called by and used (and reused) the program. In OOP, "classes" are used to group objects together and endow upon them characters that are shared.

Object Oriented Programming is more focussed on data - the program stores the data that relates to each object. However, in Process Oriented Programming, the program does not store any data and simply passes the outputs of each function to the subsequent function until the program ends. OOP considers data to be the most important, whereas POP considers the function to be the most important.

Process Oriented Programming is based on a more abstract concept of processes to operate on data structures, whereas Object Oriented Programming is based on a more tangible concept of objects to operate for a purpose. OOP therefore has a greater ability to reflect the real-world.

OOP allows for three types of accessing mode: Public, Private, and Protected to protect data, whereas POP has no such accessing modes. Further, OOP restricts data and functions to each object, whereas data in POP flows freely. Data is therefore more malleable in POP and therefore easier to use, however it also means that the data has less integrity and is less secure.

Process Oriented Programming can be more straightforward to plan and code as what needs to be thought about is the step-by-step process of how a program should run, which is more intuitive. On the other hand, Object Oriented Programming will take longer to plan as it is necessary to break-down of the program into its conceptual parts and consider the relationships within (i.e., how each object would relate to each other, parent-child classes, etc.), the data that needs to be stored (and how), and the overall functionality of the program. Diagrams will need to be drawn to demonstrate the functions and classes in the program.

However, as Process Oriented Programming follows a top-down approach, if one thing in the program needs to be changed, the other functions will also need to be changed. This is not the case in Object Oriented Programming which follows a bottom-up approach - the characteristics of each object and the result they produce are more flexible. Designs in OOP can be reused in different areas of the program whereas POP cannot. The up-front planning costs of an Object Oriented Programming model may therefore be justified where it is necessary to develop and maintain software.

As such, Object Oriented Programming is more suitable for larger-scale, more complex projects than Process Oriented Programming.

2. What's polymorphism in OOP?

Polymorphism is a core concept in Object Oriented Programming where different types (classes) of objects call the same type entity (method, operator or object) name but the type entities actually differ in how they function.

For example:

```
1  class Dog:
2      def speak(self):
3          print("Woof!")
4
5  class Cat:
6      def speak(self):
7          print("Meow!")
8
9
10
11  jeff = Dog()
12  whiskers = Cat()
13
14  jeff.speak()
15  whiskers.speak()
```

When the code is run, the following is outputted:

```
/Users/krizzle/PycharmProjects/cfg-python/venv/bin/
Woof!
Meow!

Process finished with exit code 0
```

Both objects, jeff and whiskers, are calling the same type entity name (speak()), however they come to different results. This is because jeff is calling the speak() method in class Dog, which

has different functioning (printing a different message) than the `speak()` method in class `Cat`, which `whiskers` is using.

Polymorphism allows the program to run indiscriminately with different objects that require different steps to come to the appropriate conclusion. For example, if you need every animal to make their respective noises, you would be able to simply do:

```
1  class Dog:
2      def speak(self):
3          print("Woof!")
4
5  class Cat:
6      def speak(self):
7          print("Meow!")
8
9
10
11  jeff = Dog()
12  whiskers = Cat()
13
14  animals = [jeff, whiskers]
15
16  for animal in animals:
17      animal.speak()
18
```

```
/Users/krizzle/PycharmProjects/cfg-python/ve
Woof!
Meow!

Process finished with exit code 0
|
```

And each animal would make their respective, appropriate noises, without having to manually go through and create a different condition and result for each type of animal. e.g.,

```

1  class Dog:
2      def speak(self):
3          print("Woof!")
4
5  class Cat:
6      def speak(self):
7          print("Meow!")
8
9
10
11  jeff = Dog()
12  whiskers = Cat()
13
14  animals = [jeff, whiskers]
15
16  for animal in animals:
17      if animal.__class__.__name__ == "Dog":
18          print("Woof?")
19      elif animal.__class__.__name__ == "Cat":
20          print("Meow?")
21

```

```

/Users/krizzle/PycharmProjects/cfg-python/venv/bin/python
Woof?
Meow?

Process finished with exit code 0

```

It makes things easier if things that are essentially performing a transformation in the same way have the same name. The functions may even differ in the number of arguments or the type of arguments that they accept (and often do).

Polymorphism may be implemented in parent and child classes through abstract methods where all the child classes must have a method with that method name. For example:

```

1      import abc
2
3
4      class Animal(abc.ABC):
5          @abc.abstractmethod
6          def speak(self):
7              print("*Generic animal noise*")
8              pass
9
10
11     class Dog(Animal):
12
13     def speak(self):
14         print("Woof!")
15
16
17     class Cat(Animal):
18
19     def speak(self):
20         print("Meow!")
21
22
23     jeff = Dog()
24     whiskers = Cat()
25
26     animals = [jeff, whiskers]
27
28     for animal in animals:
29         animal.speak()
30

```

```

/Users/krizzle/PycharmProjects/cfg-python/venv
Woof!
Meow!

```

If the child class does not contain the abstract method, then an error is thrown - for example, if I change class Cat to the following:

```

17     class Cat(Animal):
18
19         def meow(self):
20             print("Meow!")
21

```

The following error is raised:

```

/Users/krizzle/PycharmProjects/cfg-python/venv/bin/python /Users/krizzle/PycharmProjects/
Traceback (most recent call last):
  File "/Users/krizzle/PycharmProjects/cfg-python/17oct.py", line 24, in <module>
    whiskers = Cat()
TypeError: Can't instantiate abstract class Cat with abstract method speak

Process finished with exit code 1

```

as the abstract method `speak()` must be implemented in the `Cat` class.

So, polymorphism is particularly useful when dealing with parent-child classes. Whilst we want the parent class to have some specific functionality to produce certain outputs, how that actually functions can depend on the type of object and that child class' characteristics. A better example may be: to calculate the area of different shapes, you will need different formulas. As such, a class `Shape` may implement the abstract method `get_area()`. The way you would calculate the area of a square would be different than to calculate the area of a circle. Example below.

```

1      import abc
2
3
4      class Shape(abc.ABC):
5          @abc.abstractmethod
6          def get_area(self):
7              pass
8
9
10     class Square(Shape):
11
12         def __init__(self, length):
13             self.length = length
14
15         def get_area(self):
16             return self.length*self.length
17
18
19     class Circle(Shape):
20
21         def __init__(self, radius):
22             self.radius = radius
23
24         def get_area(self):
25             return (self.radius*self.radius) * 3.14
26
27
28     square = Square(3)
29     circle = Circle(3)
30     print(square.get_area())
31     print(circle.get_area())

```

```

/Users/krizzle/PycharmProjects/cfg-python/venv
9|
28.26

```


3. What's inheritance in OOP?

Inheritance is a core concept in Object Oriented Programming where a class (the child class) derives its traits from another class (the parent class). Upon initialisation, the child class will have all the properties and methods that are in the parent class. The child class can, and will usually have, its own additional properties and methods. It therefore functions as a subclass to the class it 'inherits' from.

A parent class can have many child classes that inherit from it. i.e., many child classes can inherit from one parent class (hierarchical inheritance). A child class can also be a parent class where another class inherits from that child class (multilevel inheritance). A child class can also inherit from two or more parent classes (multiple inheritance). Any class can be a parent class.

Inheritance is useful as it allows for easy, clear separation of different types of objects in a class without having to rewrite code. Instead you can reuse code that has already been written that can apply to all the types of objects in the class and even build on it. This also reduces the likelihood of making a mistake when manually writing these shared functions over and over again. For example:

```
1 class CFG_Software_Student():
2
3     def __init__(self, name, sponsor, project_title, fav_software):
4         self.name = name
5         self.sponsor = sponsor
6         self.project_title = project_title
7         self.attendance = 0
8         self.fav_software = fav_software
9
10    def write_code(self):
11        print("Writing code...")
12        print("Software code.")
13
14    def attend_class(self):
15        self.attendance += 1
16
17
18 class CFG_Data_Student():
19
20     def __init__(self, name, sponsor, project_title, fav_data):
21         self.name = name
22         self.sponsor = sponsor
23         self.project_title = project_title
24         self.attendance = 0
25         self.fav_data = fav_data
26
27     def write_code(self):
28        print("Writing code...")
29        print("Data code.")
30
31    def attend_class(self):
32        self.attendance += 1
33
```

The two classes CFG_Software_Student and CFG_Data_Student share many of the same properties, except that the CFG_Software_Student has a fav_software and the CFG_Data_Student has a fav_data, and that the CFG_Software_Student writes "Software code." whilst the Data Student writes "Data code.". To reduce the repeating code, we can instead create a parent class containing all the common properties that the two can inherit from.

```

1 class CFG_Student():
2
3     def __init__(self, name, sponsor, project_title):
4         self.name = name
5         self.sponsor = sponsor
6         self.project_title = project_title
7         self.attendance = 0
8
9     def write_code(self):
10        print("Writing code...")
11
12    def attend_class(self):
13        self.attendance += 1
14
15
16 class CFG_Software_Student(CFG_Student):
17
18     def __init__(self, name, sponsor, project_title, fav_software):
19         super().__init__(name, sponsor, project_title)
20         self.fav_software = fav_software
21
22     def write_code(self):
23         super().write_code()
24         print("Software code.")
25
26
27 class CFG_Data_Student(CFG_Student):
28
29     def __init__(self, name, sponsor, project_title, fav_data):
30         super().__init__(name, sponsor, project_title)
31         self.fav_data = fav_data
32
33     def write_code(self):
34         super().write_code()
35         print("Data code.")
36

```

To make additions/changes to the properties in the parent class, the child class will use `super().[functionname]` to ensure that they are inheriting. i.e., taking that code from the parent class (also known as superclass) first and foremost. Then, any additions can be made. The `CFG_Data_Student` and `CFG_Software_Student` classes now look a lot tidier, there is minimal repeated code, and we can also make other types of Students more easily, should the need arise.

The two are able to use their unique `write_code()` and `fav data/software` properties:

```

36
37
38 Kristine = CFG_Software_Student("Kristine", "Morgan", "Spotify", "Python")
39 Kristine.write_code()
40 print(Kristine.name, Kristine.fav_software)
41 Kevin = CFG_Data_Student("Kevin", "Barclays", "Weather", "SQL")
42 Kevin.write_code()
43 print(Kevin.name, Kevin.fav_data)
44
45

```

```
Writing code...
Software code.
Kristine Python
Writing code...
Data code.
Kevin SQL
```

The child class can also override a parent class method completely and implement its own method with the same method name. For example:

```
26
27 class CFG_Data_Student(CFG_Student):
28
29     def __init__(self, name, sponsor, project_title, fav_data):
30         super().__init__(name, sponsor, project_title)
31         self.fav_data = fav_data
32
33     def write_code(self):
34         print("I'm not like other students. I write Data code.")
35
36
37 Kristine = CFG_Software_Student("Kristine", "Morgan", "Spotify", "Python")
38 Kristine.write_code()
39 print(Kristine.name, Kristine.fav_software)
40 Kevin = CFG_Data_Student("Kevin", "Barclays", "Weather", "SQL")
41 Kevin.write_code()
42 print(Kevin.name, Kevin.fav_data)
43
```

Although `CFG_Data_Student` is inheriting from `CFG_Student`, the inheritance of the function `write_code()` from the `CFG_Student` class is overridden when we write the method in `CFG_Data_Student` without explicitly calling the parent classes' method through `super().write_code()` as was done before. So now, when we run the program, the output is:

```
Writing code...
Software code.
Kristine Python
I'm not like other students. I write Data code.
Kevin SQL
```

So when we call `Kevin.write_code()`, the `print("Writing code...")` from `CFG_Student's write_code()` is ignored completely, as it has been overridden in `CFG_Data_Student`.

Inheritance is an extremely valuable concept in OOP and it is easily adaptable.

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 do it by:

To start:

1. Get/create a list of people in the office
2. Use that list to create the following tables in a database:

Person(string) <ul style="list-style-type: none">- Populated from list given from (1)- INDEX- Can't be empty	Votes_Received(int)	Voted(boolean) <ul style="list-style-type: none">- Default: FALSE
--	---------------------	---

3. The results for each instance of the program are stored in a dictionary in Python, with each person's name as a key and a boolean (True or False depending on whether or not they have been voted for, default is False - not been voted for). The booleans are reset to 0 every time the program is run.

```
voting_results = {}
```

```
for person in office_list:
```

```
    voting_results.update({person: False})
```

Then, the actual voting/program will work as follows:

4. Authentication: Users will need to first fill in an authentication page in order to vote and view the voting page. This could be done by the user logging into their work account. The person's name can be taken from their work account for the purposes of identifying them for changing the status of their Voted boolean.
5. If the Voted boolean is already TRUE (i.e., this person has already voted), then they will not be allowed to vote and they will not be shown the voting page. Otherwise, they will proceed to the next voting steps.
6. Front-end: The user is presented with a list of people and a checkboxes against each name. If the user clicks the checkbox, then the checkbox is marked with a tick.
7. The user can vote for up to three people. They can submit a vote with less than three people ticked, but if they continue trying to tick after three boxes have been ticked, nothing will happen. - done with

```
while ticked_boxes < 4:
```

```
    [voting mechanism]
```

8. The user can also uncheck boxes.
9. Front-end: A submit button is displayed at the bottom. The user can then click the submit button to finalise their choice.
10. Backend: The booleans of the people in relation to whom checked checkboxes are associated with turns to TRUE. The names of the TRUE (people who received a vote) and user_name get posted to the database.

11. Database: The people with TRUE get their "Votes Received" number increased by one. The user's 'Voted' column updates to TRUE.

```
for person in voting_results.keys():
```

```
    if voting_results[person]:
```

```
        query = f"UPDATE Voting_Table SET Votes_Received += 1 WHERE Person = '{person}'"
```

```
mycursor.execute(query)
```

5. What's the software development cycle?

A software development is the iterative process-structure that is followed to build software applications. It consists of the following steps:

1. Planning
2. Requirements
3. Design
4. Implementation
5. Testing
6. Deployment
7. Maintenance

1. Planning: This stage consists of determining the scope, purpose, feasibility and necessity of the project to be undertaken, the resources and budget that are to be allocated to the project, and a timetable with goals. All the stakeholders are involved in this process.
2. Requirements: At this stage, the business team, stakeholders, and experts come together to devise the requirements for the product. A minimum viable product spec will be produced wherein the minimum requirements for a 'working' product is defined as the core of the product. The non-technical scope and purpose are translated at this stage into technical requirements that the development team can tangibly achieve. Requirements can be split into two types: functional and non-functional. Functional requirements would be what tasks the software must perform, whereas non-functional requirements would be the general properties of the software, including its visual element, as well as any maintainability, security issues. The requirements of the project will inevitably be an ongoing conversation as the scope changes or as obstacles are encountered.
3. Design: This is where the development team will transform the requirements acquired in the previous step into a detailed system architecture. The main components of the design will be the database, the front-end, and the back-end. The database's tables and its columns as well as the relationship database design must be mapped out. The front-end user interface will be drawn out. The back-end logic will be mapped out. Design patterns will be considered and identified here to inform and speed up the development process. The product manager approves/gives feedback on whether or not the details meet the set requirements before moving on to the next step.
4. Implementation: The development team actually codes the program at this stage according to the designs agreed upon, which should streamline this process. Coding guidelines as defined by the company/product manager and as generally defined as good coding practice will be followed. The development team may utilise different tools such as compilers, debuggers, and interpreters.
5. Testing and Integration: Here, the application that is developed in stage 4 is inspected and tested, usually by a Quality Assurance team. There will be a visual inspection as well as functionality testing through unit tests, alongside integration tests, performance tests, and security tests. The aim is to try and find any bugs and issues in the code which may prevent full-functionality. If any bugs or issues are identified, this is reported back to the development team who fixes it. The program is re-tested after every 'fix'. Ideally, automated testing is in place.
6. Deployment: Once the application has passed the tests and have been approved as meeting the business needs requirements, the final application is deployed. The application is integrated into its environment and any deployment issues are solved. The application is thereafter ready for use by and accessible to the user.

7. Maintenance: With the application up and running and being used, there will still be work that needs to be done on the application; fixing residual issues/bugs found by users, and making improvements to the performance of the program. If the work done is considerable, then the SDLC process must be repeated for that work. Regression testing will be employed to ensure that the application is still full-functioning after every fix and improvement.

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

Agile and waterfall are two methodologies of project management in following the SDLC process. Waterfall is where one step must be completed absolutely before moving onto the next step, whereas Agile is where you are able to go back and forth between steps after receiving feedback. Waterfall is therefore a lot more structured than Agile, whereas Agile is more adaptive and flexible.

Waterfall is useful for particularly sensitive, delicate processes where if the product does not meet the full requirements, then that would be disastrous, e.g., aerospace systems, as the expectations and deliverables of each stage are clear and required.

Agile is useful where the requirements for a product can and will be subject to change depending on external circumstances or if a new product is being developed, in respect to which the specs are not yet clear and will become more clear with further testing, development, and research. Agile also acknowledges that technology can change significantly during the development of a product. On the other hand, Waterfall is best suited for projects with a clear end goal, where the project owner has a definitive vision for the project and is confident in it.

Agile will not be the best choice if extensive documentation is needed for the product, for example if there is a high level of regulation for that type of product. This is because the product, its functions and its requirements are constantly changing such that it does not lend itself to be documented easily. On the other hand, Waterfall is great for this purpose as everything is done thoroughly and methodically.

Agile projects are usually broken down into Sprints, where there is a continuous, structured, conversation going on between the product manager and the development team, as well as with the stakeholders. The team works on small phases of the project in each Sprint with a short-term deadline, usually 2 weeks. On the other hand, in Waterfall, the requirements and expectations and timeline are defined up-front. As a result, ongoing discussions with all the stakeholders is not a part of Waterfall - Waterfall is rather a more hands-off approach.

Waterfall projects may take longer to complete as each phase needs to be completed in its entirety before progressing, whilst Agile allows different phases of the project to be worked on at the same time. However, Agile may take longer as the expectations and requirements are not clearly defined and there will be unnecessary effort spent due to potential overlap or where something needs to be later modified.

7. What is a reduced function used for?

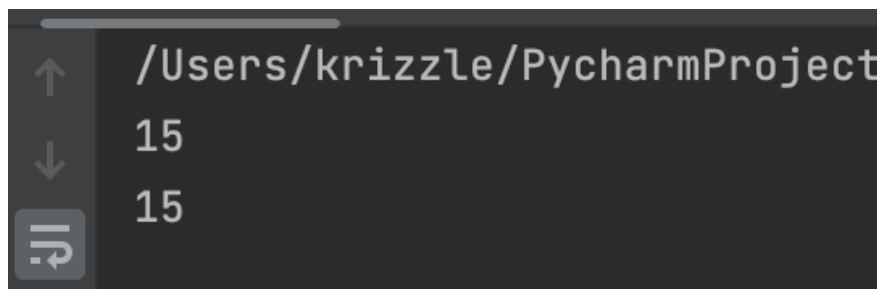
A reduce function is able to take a function and an iterable and apply that function on a rolling basis across that iterable such that the output is a single value. The function can be a lambda function or the name of another defined function. The reduce function must be imported from the Python in-built functools module.

For example,

```
1  from functools import reduce
2
3  def func(x,y):
4      return x+y
5
6  lst = [2,5,1,3,4]
7
8  print(reduce(func, lst))
9
10 print(reduce(lambda x,y: x+y, lst))
11
```

Line 8 demonstrates a previously defined function being passed as an argument, whereas Line 10 demonstrates using a lambda function as the argument.

This code prints:



The image shows a terminal window with the following output:

```
/Users/krizzle/PycharmProject
15
15
```

Both reduce functions come to the same result.

Reduce can also take a third argument which will be the first value passed to the lambda function as its initialiser (and then reduce iterates through the list). For example:

```

1  from functools import reduce
2
3  def func(x,y):
4      return x+y
5
6  lst = [2,5,1,3,4]
7
8  print(reduce(func, lst, 5))
9
10 print(reduce(lambda x,y: x+y, lst))

```

The two prints are now:

```

/Users/krizzle/PycharmP
20
15

```

Where the first iteration is $5 + 2$, then $7 + 5$, then $12 + 1$, etc. The default value of the initialiser is None.

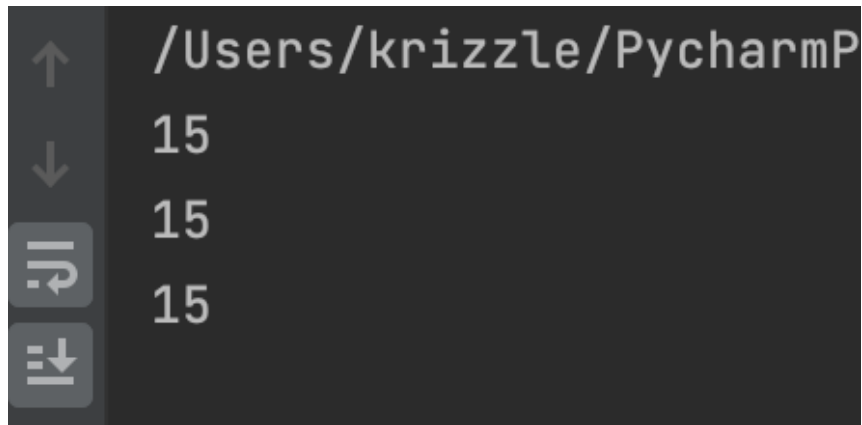
The reduce function works in a similar way to a for-loop - it performs the function on first two elements of the iterable, and then takes the result and performs the function with the result and the third element, then so on. In the case of summing up two numbers, as above, the for loop would be written as follows:

```

14  for i in lst:
15      total += i
16
17  print(total)
18

```

And the output is:



```
↑ /Users/krizzle/PycharmP  
↓ 15  
↺ 15  
⇅ 15
```

They both come to the same result, and a for-loop would arguably be easier to read, however using reduce is more concise and performs better. The reduce function was moved from an automatic in-built function to a function in the functools library that needed to be imported, as the author of Python advises minimal use of the functools (and instead for programmers to use a for-loop. Its use, therefore, should be limited to situations where the performance of the code would be drastically improved by using a reduce function (i.e., huge, complex function/list).

8. How does merge sort work

A merge sort follows a Divide and Conquer approach to sorting a list.

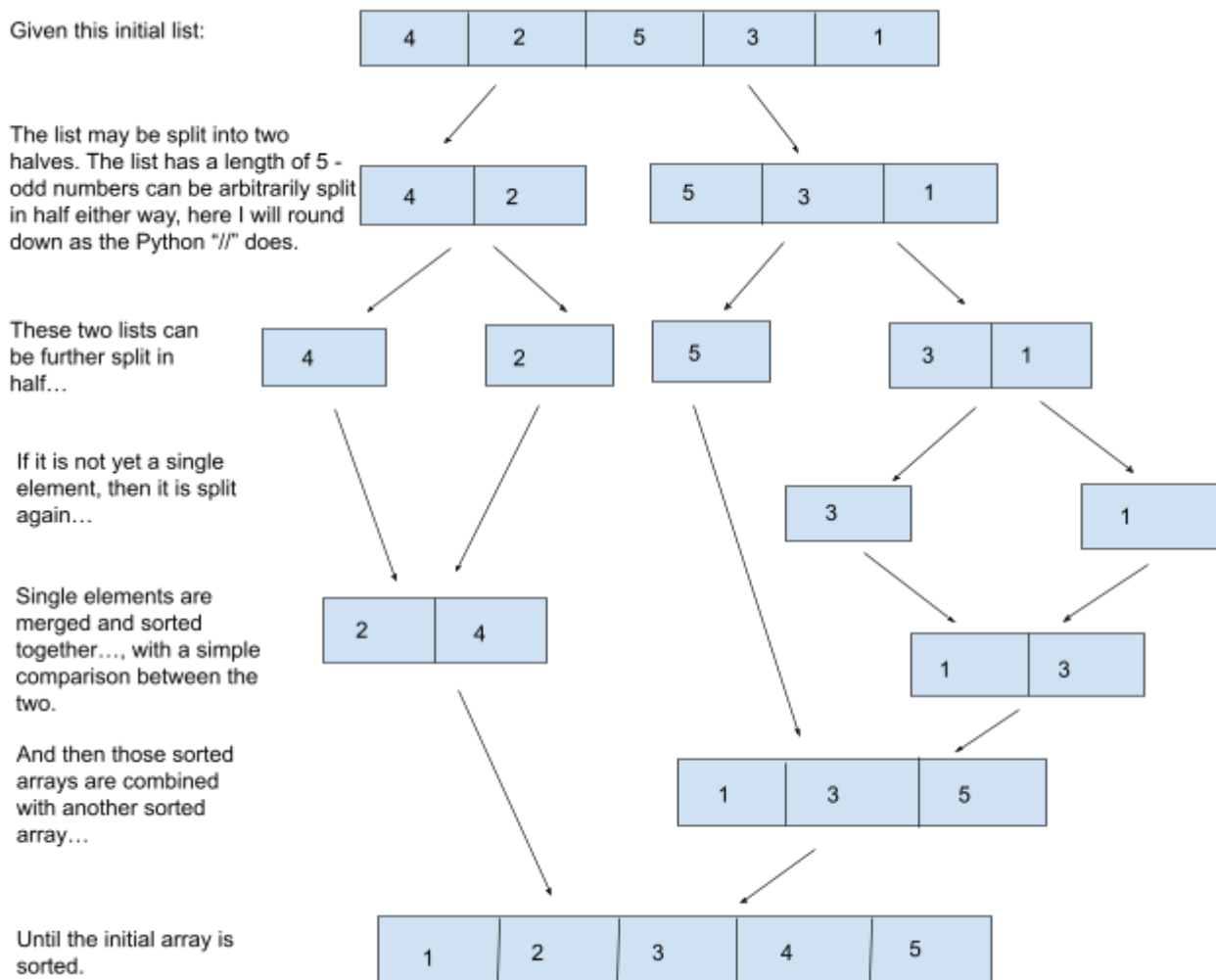
First, the problem is separated into sub-problems. That is, the given array is split in half over and over again until only single-elements remain in each branch. This is usually done with a recursive function.

Then, the single-element arrays are combined into a two-element array, having been sorted between them.

Then, these two-element arrays are combined with another two-element array, having been sorted between them. And so on, until the original given array has been fully sorted.

The merge sort is more efficient than. e.g., a bubble sort that goes through each element of the array and compares itself with the subsequent element to sort, for longer lists, as it can safely assume that for. e.g., two lists with 4 elements each, that: if element 1 in list B is greater than elements 1, 2, and 3 in list A, then the later elements 2, 3, and 4 in list B must also be greater than elements 1, 2, and 3 in list A, as list A and B are already sorted. That is, the Divide and Conquer technique reduces the need to compare between every single element as the sub-arrays have already been sorted.

A merge sort can be visualised as follows:



In the step where the array A [2,4] is to be combined with array B [1,3,5] to create a resulting array C, the logic is applied:

1. Compare index 0 of array A (2) and array B (1). 1 is less than 2, so 1 is the first element of array C.
2. Compare index 0 of array A (2) and index 1 of array B (3). 3 is greater than 2, so 2 is the next element of array C.
3. Compare index 1 of array A (4) and index 1 of array B (3). 3 is less than 4, so 3 is the next element of array C.
4. Compare index 1 of array A (4) and index 2 of array B (5). 5 is greater than 4, so 4 is the next element of array C.
5. The final remaining element 5 is added.
6. The resulting array C is [1,2,3,4,5].

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?

Generator functions are useful as they do not automatically give you the full set of results outright. Rather, with a Generator function, you can choose how many results you actually want (if you don't know already), and you can get the results one-by-one. The benefits of both are discussed below.

One-by one

Generator functions will give you the next value of the iterator when called, such that you receive the result values one-by-one as they are required, rather than altogether at once as a normal function that returns all the results would do. This means, that you are able to process the results and deal with them separately, as you might do in any case, with the added benefit that you do not need to allocate the memory for all the results at one time. That is, you are able to perform 'Lazy Evaluation', where you perform an evaluation if and when it is needed. Indeed, if you get all your results at once and too much memory is consumed at once, your server might crash - the maximum size of a Python list on a 32 bit system is 536,870,912 elements.

A valuable use case for this would be if you are importing and processing rows of a large table in a database. For example, you might want to contact all users of your app via email. If you have a very successful app with over 1B users, then if you extract all the user's emails from your database with a return function, then this could crash your server. Instead, you could use a generator function to give you 500k users' emails at a time, and you would work through those users and send them the emails, and then move on to the next 500k, and so on. The generator also updates with the latest information when `next()` is used. That is, if the user had not signed up at the time that the generator function was first called, they will still be caught by the later gathering of email addresses. So generator functions are also useful where it is important to ensure that the program is as up to date with the newest information as possible.

Partial results

With a generator, you are able to extend your list of results and stop where this is no longer useful for you any longer. For example, [example]. Therefore, with a generator, you are able to create and use an infinite list without crashing the program. For example:

```
1 def gen():
2     i = 0
3     while True:
4         i += 1
5         yield i
6
7     gen = gen()
8
9     print(next(gen))
10    print(next(gen))
11    print(next(gen))
12    print(next(gen))
13    print(next(gen))
```

```
↑ /Users/krizzle/PycharmProjects/c
↓ 1
⏮ 2
⏪ 3
⏩ 4
⏭ 5
⏴
```

The above generator returns consecutive numbers each time `next()` is called. It does not have an end-condition, as would be required from a function. Instead, it can generate an infinite amount of numbers, and this will be useful where it is not certain at what point you want to stop getting numbers/results (or if you might want them for a long time!). An example use case could be assigning unique number IDs to employees. It is not certain how many employees you will employ, and as such this will be an easy, quick way to generate a unique number.

However, a generator function has the disadvantage of having to re-run your generator if you want to reuse a previous element. Further, whilst a generator function helps you reduce the memory needed upfront for your results, you will need to keep in memory the variables within the generator function which may be costly.

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

A decorator is a function that takes another function as an argument and returns a modified, extended version of that function. Rather than being called like a traditional function, it can be added directly to the function by using "@<decorator_function_name>".

Timing

One idea for a useful decorator is one that you can add to functions to help you time how long they take to execute. We can do this by:

```
1  from time import time
2
3  def timer(func):
4      def inner(*args, **kwargs):
5          start_time = time()
6          result = func(*args, **kwargs)
7          end_time = time()
8          time_elapsed = (end_time - start_time) * 1000
9          print(f"This took {time_elapsed}ms to run")
10         return result
11     return inner
```

The `time.time()` function returns the time since epoch in seconds. This decorator uses `time()`, and then calls the function as it would be called in the program, then uses `time()` again, and then takes away the final time from the beginning time to get the total time elapsed. Finally, it returns the output received from calling the original function without modifying it. The time elapsed is multiplied by 1000 to get the time in ms rather than s, as most functions run very fast!

We can time these functions:

```
14 def addition(x, y):
15     return x+y
16
17
18 def sum_nums(num_lst):
19     total = 0
20     for num in num_lst:
21         total = total + num
22     return total
23
24     nums = [x for x in range(10000)]
25
26     print(addition(2,3))
27     print(sum_nums(nums))
28
```

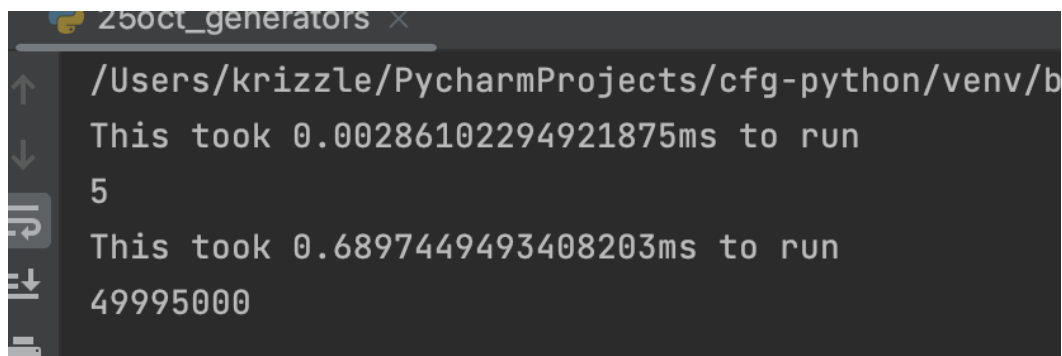
```
↑ /Users/krizzle/Pyo
↓ 5
↺ 49995000
```

To:

```

13     @timer
14     def addition(x, y):
15         return x+y
16
17     @timer
18     def sum_nums(num_lst):
19         total = 0
20         for num in num_lst:
21             total = total + num
22         return total
23
24     nums = [x for x in range(10000)]
25
26     print(addition(2,3))
27     print(sum_nums(nums))
28

```



```

25oct_generators x
/Users/krizzle/PycharmProjects/cfg-python/venv/b
This took 0.00286102294921875ms to run
5
This took 0.6897449493408203ms to run
49995000

```

No amendments need to be directly made to the functions themselves, which means that the timing can also be taken away from the functions easily, which is a strength of keeping this timer function in a decorator. You might want to time it a couple times when testing, but remove it when it comes to the fully-functioning product.

Authorisation

You may also use a decorator to ensure that a user is logged in or authorised in order to perform a certain action/function. This decorator may be written as follows:

```

1  def auth_required(func):
2      def inner(user, *args, **kwargs):
3          if not user.logged_in:
4              print("Please log in.")
5              return ""
6          elif user.logged_in:
7              return func(*args, **kwargs)
8      return inner
9

```

If the user is not logged in, then the program asks the user to log in and returns nothing, otherwise the function is returned as normal. I have written a user class to accompany this example.

```

8
9  class User():
10     def __init__(self, name, logged_in):
11         self.name = name
12         self.logged_in = logged_in
13
14     def get_name(self):
15         return self.name
16
17     def logged_in(self):
18         return self.logged_in
19
20     def access_sensitive_data(self):
21         return "Accessing very sensitive data."
22

```

Where logged_in returns a boolean, depending on whether or not the user is logged in. If this is used without a decorator...

```

23
24  Kristine = User("Kristine", False)
25
26  print(Kristine.access_sensitive_data())
27

```

```

↑ /Users/krizzle/PycharmProjects/cfg-pyth
↓ Accessing very sensitive data.

```

A not-logged-in user can access very sensitive data! However, if we add the auth_required decorator to access_sensitive_data(), then:

```

8
9  class User():
10     def __init__(self, name, logged_in):
11         self.name = name
12         self.logged_in = logged_in
13
14     def get_name(self):
15         return self.name
16
17     def logged_in(self):
18         return self.logged_in
19
20     @auth_required
21     def access_sensitive_data(self):
22         return "Accessing very sensitive data."
23

```

```

25
26
27  Kristine = User("Kristine", False)
28
29  print(Kristine.access_sensitive_data())
30

```

```

/Users/krizzle/Pycharm
Please log in.

```

Not-logged-in user Kristine can no longer access the sensitive data, and is instead told to log in. On the other hand:

```

26
27  Kristine = User("Kristine", False)
28  David = User("David", True)
29
30  print(Kristine.access_sensitive_data())
31  print(David.access_sensitive_data())
32

```

```
↑ /Users/krizzle/PycharmProjects/cfg-pyt  
↓ Please log in.  
↺  
↻ Accessing very sensitive data.
```

Logged-in user David can access the very sensitive data!

The decorator allows us to add this condition to many functions without having to rewrite it over and over it, reducing repetition. There will be many uses for when a user would have to be logged in in order to e.g., access certain webpages, or e.g., be able to perform certain functions. e.g., save a picture.

Abusive

An abusive decorator would be one that changes the meaning and/or return value of the original function such that the code is not easily understandable and it is not clear how or why the output is what it is.

For example:

```
generators.py × generators_2.py × abusive_decorator.py × 25oct_
1 def month_or_year(func):
2     def inner(*args, **kwargs):
3         result = func(*args, **kwargs)
4         if result > 100:
5             return f"I am {result} years old"
6         else:
7             return f"I am {result} months old"
8     return inner
```

This abusive decorator checks if the number output from a given function call is above 100. If it is, then it returns a string interpreting the result as years, and if it is not, then it returns a string interpreting the result as months.

If we return back to our `addition()` and `sum()` functions and add the `month_or_year()` decorator now..

```

py 11     @month_or_year
.py 12     def addition(x, y):
13         return x+y
14
15     @month_or_year
16     def sum_nums(num_lst):
17         total = 0
18         for num in num_lst:
19             total = total + num
20         return total
21
22     nums = [x for x in range(10000)]
23
24     print(addition(2,3))
25     print(sum_nums(nums))
26

```

```

n:  abusive_decorator x
  /Users/krizzle/PycharmProjects/cfg-pyth
  /Users/krizzle/PycharmProjects/cfg-pyt
  I am 5 months old
  I am 49995000 years old
  Process finished with exit code 0

```

When we look at what the original functions are doing (and even the decorator), it is not clear why the output is that. Furthermore, the output of the function is changed from an integer to a string, which makes the code confusing and unreadable. The name of the function is also not explicit in its purpose.