Code First Girls Nanodegree—Software Module Theory Assignment

Robyn Seymour-Jones

Contents

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

Programming paradigms are different programming styles that implement different strategies and structure code in different ways. Object-oriented and process-oriented are two such programming paradigms and take contrasting approaches to structuring a program. The fundamental difference between the two is that process-oriented programming separates the data from the processes that can be performed upon the data while object-oriented programming (OOP) groups both the data and the processes together.

Depending on the programming language being used, it is often possible to implement either paradigm. For example, either approach can be implemented with Python. Other languages that support OOP include C, C++, Java, Ruby and Go. The decision over which paradigm to use depends on the program you are trying to create.

Process-oriented (i.e. procedural) programming

Procedural programming is often the first programming paradigm that a new developer will learn and had been the type of programming we were doing on this course prior to learning about object-oriented programming. In procedural programming, a program is written as an ordered series of steps that are executed one after the other. It focuses on the steps that are taken to complete a task. It is usually divided up into smaller parts called procedures, routines or functions. Data are treated as separate entities to these functions.

Advantages:

- Procedural programming is simple, straight-forward and very good for general purpose programming.
- The linear structure means that it is easy to track the flow of the program.
- By saving blocks of reusable code as functions, the code can be easily reused in different parts of the program.

Disadvantages:

- The data is stored openly within the program so is exposed to the whole program, which is not the most secure approach.
- It can be difficult to relate the data and processes to real world objects.

Object-oriented programming

In OOP, data and the processes that can be performed upon the data are grouped together into objects. Each object belongs to a class, which is effectively a blueprint for an object. The class specifies the attributes that an object of that class will have (i.e. the description of the object) as well as the methods that can be performed upon on objects of that class (i.e. the actions that can be performed on the object). When an object of a class is created, the object is referred to as being an instance of the class and the class is said to be instantiated.

There are four key principles of OOP, two of which (inheritance and polymorphism) are covered in more detail in the next two questions of this assignment.

- Inheritance—classes can inherit attributes and methods from other classes.
- Polymorphism—the ability of a method to perform differently depending on the class of the object it is acting on.
- Encapsulation—the bundling of data and methods acting on the data into a single unit (object), which puts restrictions on how data and methods can be accessed and prevents accidental modification.
- Abstraction—the hiding of internal implementation details.

Advantages:

- Suitable for large, complex programs as the modularity of object-oriented programs makes them easier to maintain than procedural programs.

- Good for collaboration as different developers can each work on separate parts of the program without impacting the others' work.
- The reusability of the code can make development faster.

Disadvantages:

- Steeper learning curve than procedural programming.
- Programs are larger with more lines of code required than in procedural programming. This can also mean that they are slower.

Code comparison

Two code examples are shown below that both store data about some cats and print out some information about them. The following is what is outputted by both programs:

```
'Meow, meeooooooww!!'
'Gertrude is a 3 year old ginger cat.'
'Kenneth is a 5 year old black and white cat.'
```

This first program uses procedural programming to achieve the output. Data are stored to variables and procedures are stored as functions for easy recall later.

```
# PRODECURAL PROGRAMMING EXAMPLE
# ——————————————————————————

# Store data to variables for reuse later.
cat1_name = "Gertrude"
cat1_age = 3
cat1_colour = "ginger"

cat2_name = "Kenneth"
cat2_age = 5
cat2_colour = "black and white"


# Create meow function.
def meow():
    print("Meow, meeooooooww!!")


# Create info function.
def get_info(name, age, colour):
    print(f"{self.name} is a {self.age} year old {self.colour} cat.")


# Create program flow.
meow()
get_info(cat1_name, cat1_age, cat1_colour)
get_info(cat2_name, cat2_age, cat2_colour)
```

This second program uses OOP to achieve the same output.

```
# OOP EXAMPLE
# ——————————

class Cat:

    def __init__(self, name, age, colour):
        self.name = name
        self.age = age
        self.colour = colour)

    def meow(self):
        print("Meow, meeooooooww!!")

    def purr(self):
        print("Puurrrrrr")

    def get_info(self):
        print(f"{self.name} is a {self.age} year old {self.colour} cat.")


# Instantiate Cat class (create objects of class Cat).
cat1 = Cat("Gertrude", 3, "ginger")
cat2 = Cat("Kenneth", 5 "black and white")


# Call instance methods on instances of Cat class.
cat1.meow()  # 'Meow, meeooooooww!!'
cat1.get_info()  # 'Gertrude is a 3 year old ginger cat.'
cat2.get_info()  # 'Kenneth is a 5 year old black and white cat.'
```

2 | What's polymorphism in OOP?

Polymorphism is one of the four main concepts in object-oriented programming (OOP) along with inheritance, abstraction and encapsulation. It literally means "having many forms". In OOP, polymorphism refers to the ability of a function, method or operator to perform different actions depending on the class of the object that it is acting on. A very simple example of this is how the `+` operator will behave differently depending on whether the objects it is acting on are integers or strings:

```
>>> 2 + 3
5

>>> "2" + "3"
'23'
```

Polymorphism allows us to define different methods in different classes but with the same name. In the following example, calling the `caffeine` method will perform different actions (in this case it will print a different statement) depending on the class of the object it is acting on. (I am working in a coffee shop at the moment, which is how I ended up with this example!)

```
>>> class Coffee:
...     def caffeine(self):
...         print("I contain 70 mg of caffeine.")

>>> class Tea:
...     def caffeine(self):
...         print("I contain 35 mg of caffeine.")

>>> latte = Coffee()
>>> earl_grey = Tea()

>>> latte.caffeine()
'I contain 70 mg of caffeine.'

>>> earl_grey.caffeine()
' I contain 35 mg of caffeine.'
```

Implementation – overloading

Polymorphism can be implemented with **method overloading** or **method overriding**. Both of the code examples provided in this answer so far are examples of implementing polymorphism through overloading (the first example used operator overloading and second example used method overloading). Method overloading is when multiple methods exist with the same name but different

definitions. The method that runs when a method with that shared name is called will depend on the class of the object in it acting on (as in the two examples above) or on the arguments provided. In terms of the latter, some languages will let you define multiple methods with the same name within a class but that have different paraments. Python doesn't let you do this, but you can still work around it to implement polymorphism this way. The example we did in class shows how this can be done. In the code below, if one argument is provided the method will perform in a certain way (it will return the number provided as an argument) and if two arguments are provided it will perform in a different way (it will sum the provided arguments):

```
>>> class Car:
...     def start(self, a, b=None):
...         if b is not None:
...             print(a + b)
...         else:
...             print(a)

>>> mini = Car()
>>> mini.start(2)
2
>>> mini.start(2, 3)
5
```

Implementation – overriding

Polymorphism is useful when implementing inheritance in OOP. A child class inherits the methods of its parent class (discussed further in the next questions). As shown in the example code that follows, polymorphism allows these inherited methods to be overridden:

```
>>> class Drink:
...     def caffeine(self):
...         print("I don't contain caffeine.")

>>> class Coffee(Drink):
...     def caffeine(self):
...         print("I contain 70 mg of caffeine.")

>>> class Tea(Drink):
...     def caffeine(self):
...         print("I contain 35 mg of caffeine.")

>>> water = Drink()
>>> latte = Coffee()
>>> earl_grey = Tea()

>>> water.caffeine()
'I don't contain caffeine.'

>>> latte.caffeine()
'I contain 70 mg of caffeine.'
```

```
>>> earl_grey.caffeine()
' I contain 35 mg of caffeine.'
```

3 | What's inheritance in OOP?

Within object-oriented programming (OOP), inheritance is the ability of a class to take on (usually termed "inherit", "derive" or "extend") the attributes and methods of another class. The class that the attributes and methods are inherited from is known as the **base, parent or super class**. The class that inherits the attributes and methods is known as the **derived, child or sub class**.

The relationship between the parent and child class is said to be an "is a" relationship where the child class **is a** specialised version of the parent class. This means that the child class can be substituted for the parent class in a program without disrupting the behaviour of the program (this is the Liskov substitution principle).

The child class can extend the attributes and methods of the parent class. It can also override the attributes and methods of a parent class (this was explained in the previous answer on polymorphism so is not covered here any further).

The following example demonstrates how a child class can inherit the method of its parent class. `vehicle_method()` can act on an object of class `Car` as it inherited this method from the parent class `Vehicle`. It also demonstrates how a class can be extended with the addition of another method in the child class `Car` that cannot be accessed by an object of the parent class.

```
# Create Class Vehicle
>>> class Vehicle:
...     def vehicle_method(self):
...         print("I am a vehicle that gets you from A to B.")


# Create Class Car that inherits from Vehicle class
>>> class Car(Vehicle):
...     def car_method(self):
...         print("I am a car, a type of vehicle.")


# Create and check a general vehicle object.
>>> general_vehicle = Vehicle()
>>> general_vehicle.vehicle_method()
'I am a vehicle that gets you from A to B.'


# Create and check a car object.
>>> my_car = Car()
>>> my_car.vehicle_method()
'I am a vehicle that gets you from A to B.'
>>> my_car.car_method()
'I am a car, a type of vehicle.'
```

```
# Demonstrate that child method does not work on parent
>>> general_vehicle.car_method()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Vehicle' object has no attribute 'car_method'
```

Multiple parents and child classes

A parent class can have multiple child classes and a child class can have multiple parent classes. A child class can subsequently have its own child classes. In this way, a hierarchy of classes can be set up.

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?

NOTE: This question doesn't state how we should be answering it or what aspect of the course we should be demonstrating knowledge from. I'm going to assume it is testing our knowledge of process flow and system design as we were asked a similar question during that lesson. We have already had several questions on OOP so I'm guessing it doesn't want us to consider how we would implement it that way, although I know this would be one approach to it. It is a bit confusing as this is the software theory assignment but process flow and system design were covered in the foundation course. However, the SDLC and agile were also part of the foundation syllabus rather than the software syllabus and there are questions on those here, so I will assume it is ok to talk about process flow and system design. Also, I am assuming it is not testing our coding ability as this is a theory assignment and not a practical one.

First, I would list out all of the key requirements and deliberate over the considerations to be made. I would then decide which technologies and approaches are most suitable. Finally I would create a process flow diagram to illustrate the steps that a user moves through when using the app. This would help me to see more clearly which functions I would need to write. The following shows how I would go about the above for this specific voting app.

Requirements

- It must have a front end so the user can interact with the system. This front end could be as simple as being in the command-line or it could be more fancy using something like flask.
- It must have a user input to allow the voter to enter who they want to vote for into the system. It should list the names of these people and the user selects which three they want to vote for, rather than a free text input in case that results in missed votes through things like spelling mistakes. It should make sure they select three unique individuals rather than vote for one person three times.
- It must have a database with a table that stores the number of votes for each "funny" person.
- It must allow someone to vote only once. This would require some sort of voter registration so that each person was assigned a unique voter ID number.

- It must have a table in the database that stores a record of voter ids that have voted (so they can't vote again).
- Votes must only be submitted within a given time window.
- There must be a way to calculate and view the results once voting has closed.

Considerations

- This doesn't seem like a serious vote so it probably won't require backup systems or plans in place in case the internet isn't available. But these are things that would need to be considered for more serious voting systems.
- Should the user be able to rank their votes and should the system take this into consideration or should they just vote for three people without ranking?

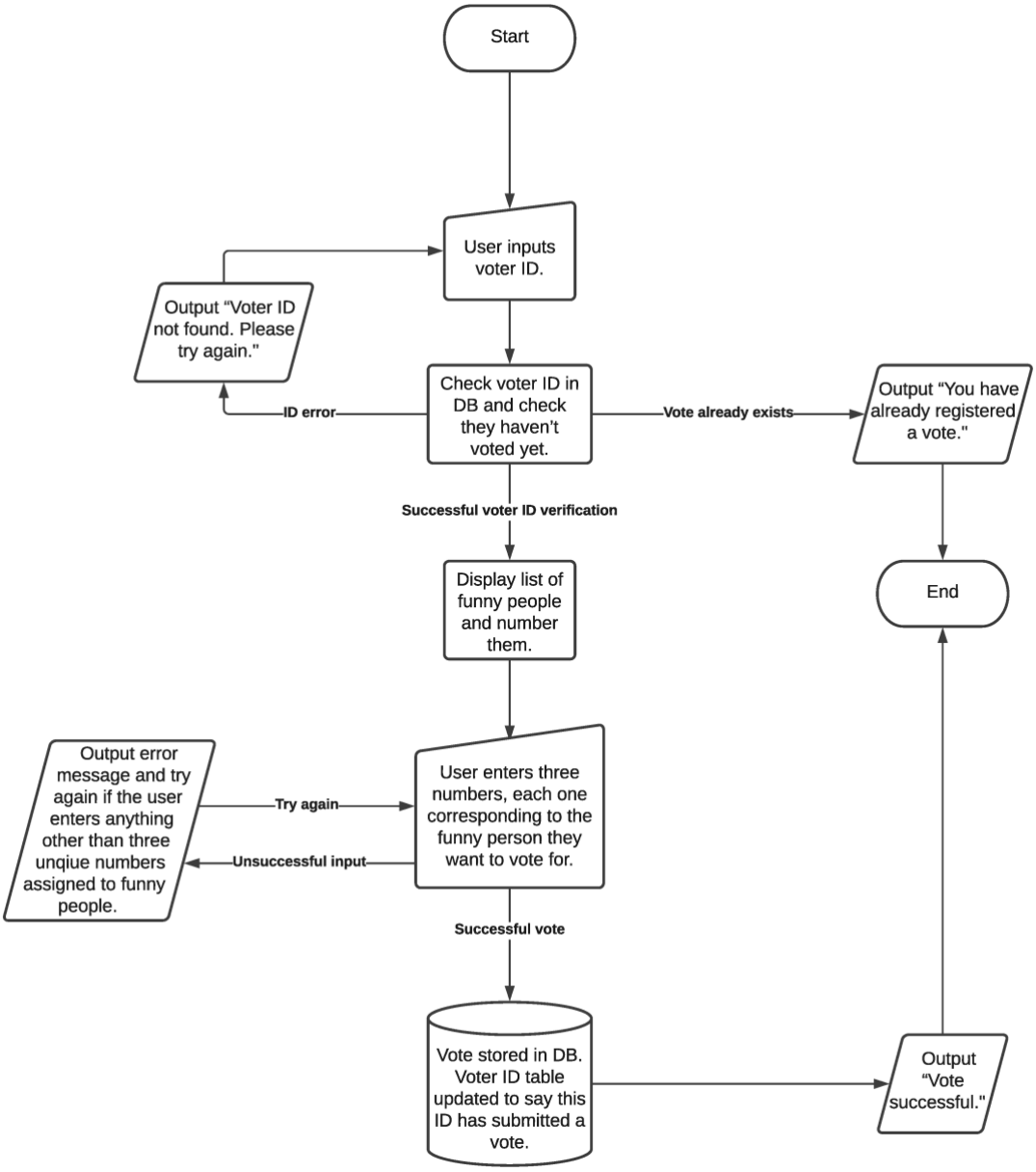
Technologies and tools

- MySQL database with a table to store votes cast for each funny person, a table to store voter registration details and whether they have already voted or not.
- Backend logic with Python. This would include:
    - a server-side API app
    - a database connection and functions that retrieve data and write to the database
    - a client-side script with:
        - a function that accepts user input of voting ID and checks if they are registered and if they have voted yet
        - a function that accepts user input of the three unique people they are voting for and saves this data to the database as well as marking the ID down as having cast a vote

- A simple front-end interface.

Process Flow Diagram

Note that this is just an example and only shows the user making a vote. No registration process or accessing of results is demonstrated here.



5 | What's the software development cycle?

The software (sometimes system or application) development life cycle (SDLC) is the structured framework of stages that are worked through when a piece of software is created (Figure 5.1). It forms the basis for project planning and scheduling. Following the SDLC helps to ensure that the highest quality software is developed at the lowest cost and in the shortest time possible.

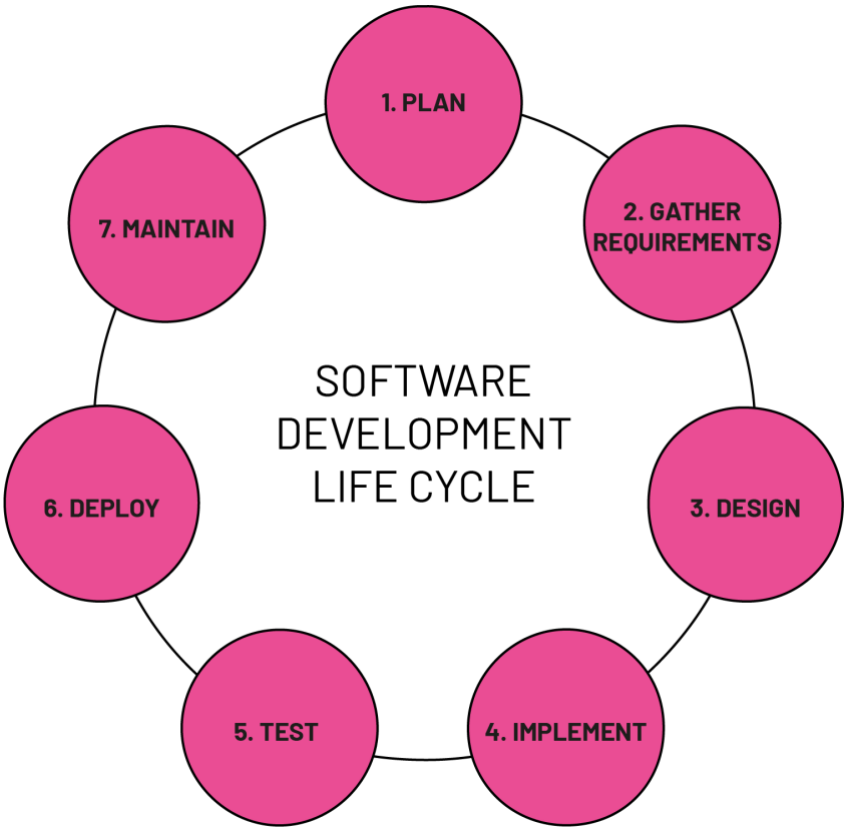


Figure 5.1—Diagram showing the SDLC stages.

Stages of the SDLC

The cycle has several stages, starting with the planning stage and proceeding through the requirement gathering, design, implementation, testing, deployment and maintenance stages:

1. **Plan**. During the planning stage, the problem the software is going to solve is defined. The working methodology will be decided upon (e.g. agile or waterfall, which are discussed in the next theory question). Funding and resources are acquired.

2. **Gather requirements and analysis**. Thorough research is carried out to understand the needs of the end users and identify the specific tasks the software will need to carry out. Existing solutions to similar problems are researched. A feasibility study is carried out to make sure the software can be complete within time and in budget. A software requirement specification (SRS) document is drawn up documenting all of these things.

3. **Design**. The specifications, features and operations that the software will require are planned out. This includes high-level design such as planning out the overall system architecture and outlining the role of each module/script. It also includes low-level design such as planning the database base tables and outlining the functional logic of each modules/script.

4. **Implement**. This is usually the longest stage in the SDLC. It is the stage where the code is written. Larger tasks are divided up into smaller chunks and assigned to different developers.

5. **Test**. During this stage, the software is tested to ensure it works as required and to identify any bugs. These need to be tracked, fixed and retested.

6. **Deploy**. The software is released into a real-world environment where real users can make use the system.

7. **Maintain**. This is an ongoing stage to ensure that the software continues to meet the needs of the end users. End users might provide feedback on their experience of using the software. Upgrades and bug fixes are implemented.

SDLC Models

There are different working models that make use of the SDLC in different ways. Two of the most popular—Waterfall and Agile—are discussed in the next question.

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

Agile and Waterfall are different approaches to project management. They both follow the software development life cycle (SDLC) but in different ways.

Waterfall Methodology

Waterfall is the traditional model for software development. It is a linear, sequential approach with each phase of the SDLC being worked through one at a time. Each phase is completed before the next one begins. It means that the design is thoroughly thought out and planned before coding begins, which is known as big design up front (BDUF). The first time that a complete working piece of software is created is near the end of the development process. This piece of software should theoretically be very close to the final working version that is deployed.

A waterfall approach can result in potential problems being identified early, which can reduce the costs of later stages. It also tends to lead to better documentation with detailed requirements and design documents created in order for the subsequent stages of the development cycle to proceed. The main problem of this approach is that it does not allow for much flexibility or consultation with clients throughout the process. The client must know the full details of what they want before the development process starts and they won't be able to change their minds on parts of the design after development has started.

Agile Methodology

The agile model for software development is a more recent concept than the waterfall model but has risen rapidly in popularity. It is an iterative, incremental approach consisting of several shorter development cycles (sprints) that each aim to improve on the product. A list of tasks required to create the product is drawn up and items are prioritised. This is the product backlog. Tasks from the product backlog are selected based on priority for each sprint, creating the sprint backlog. These tasks are worked on resulting in an increment of progress being made towards the final product.

Working prototypes of a product can be created quite early on in the development processes, allowing for clients to provide feedback and to request changes to the design. This flexibility and adaptability are the key features of the agile approach. It is all about putting the client first and making sure they get the outcomes they are after, with the developers being able to accommodate changes at any stage of development. The agile approach prioritises working software over documentation, meaning documentation is not as rigorous as it is for software developed under a waterfall methodology.

The agile approach is not as intuitive and straightforward as the linear approach of waterfall. Different agile frameworks have been created to help implement this methodology. An example is the Scrum framework.

7 | What is a reduced function used for?

Reduction is the process of converting an expression into a simpler form, e.g. `5+(3*4)` → `5+12`. In programming, it usually refers to reducing the values of an array into a single result.

Functional programming

Reduction is a key aspect of functional programming. This is a programming paradigm (see answer to question 1) in which data flow from one function to the next. The output of one function becomes the input for another function. Functions are self-contained. They do not have side-effects, which means they do not update/modify any global variables or objects. They produce an output that depends only on the input.

There are three functions that facilitate functional programming in Python – `reduce()`, `map()` and `filter()`.

The reduce function

The `functools` module contains a function called `reduce()`. This takes a function (one with two arguments) and an iterable as arguments then applies the function to each element of the iterable cumulatively (one argument being the cumulative value so far, the other being the next element in the iterable) before returning the final cumulative value, e.g.

```
>>> from functools import reduce

>>> lst = [1, 2, 3, 4, 5]

>>> def func(a, b):
...     return a*b

>>> product_of_lst = reduce(func, lst)
>>> product_of_lst
120
```

The Python documentation says the `reduce()` function is similar to the following Python function:

```
def reduce(function, iterable, initializer=None):

    # dealing with optional initializer value
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer

    # main part of function
    for element in it:
        value = function(value, element)
    return value
```

Use cases

The `reduce()` function is not seen as being very Pythonic. It has been demoted from the main Python library into the `functools` library and other functions now provide some of its functionality (e.g. `sum()`, `max()`, `min()`). It is still around because of its popularity with functional programmers. You would use it if you were trying to implement functional programming. You might also use it for time efficiency reasons. The `reduce()` function can be faster than the Python `for` loop as it (and therefore its internal looping) is written in C.

One of the most common use cases for `reduce()` is as an accumulator (i.e. the sum of all values in a list). However, since `sum()` does the same thing and is part of the main Python library, it is now the preferred function in this use case. Some sorting algorithms use reductions to be able to handle very large datasets.

8 | How does merge sort work?

Merge sort is a sorting algorithm that uses the divide and conquer approach. This approach splits the problem (an unsorted array) into smaller problems (multiple smaller unsorted arrays) that are easier to solve. It is one of the most efficient sorting algorithms with a time complexity of O(nlogn) and a space complexity of O(n).

Divide

The divide stage consists of splitting an unsorted array in half, down the middle, into smaller sub-arrays. This process is repeated on each sub-array and continues on all resulting sub-arrays generated until the individual elements are isolated (Figure 8.1).

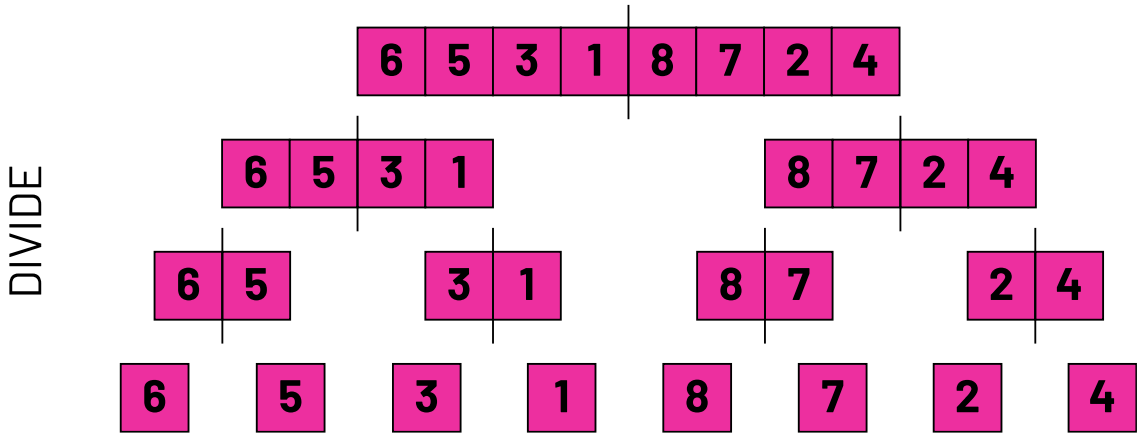


Figure 8.1—Diagram demonstrating the divide portion of the merge sort algorithm.

Conquer

The conquer, or merge, stage of the merge sort algorithm evaluates each group of numbers that previously made up a sub-array and recreates that sub-array but in ascending numeric order (Figure 8.2).

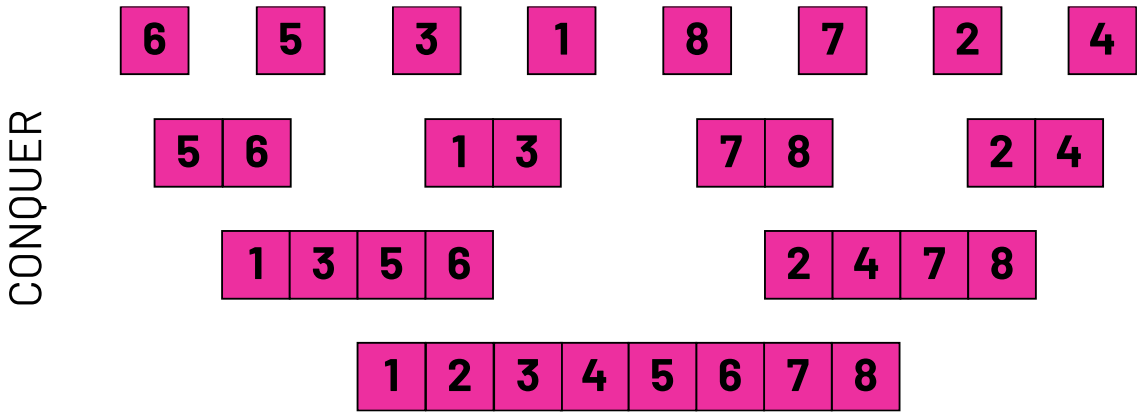


Figure 8.2—Diagram demonstrating the conquer (merge) portion of the merge sort algorithm.

Implementation in Python

Implementing the merge sort algorithm in Python involves recursion. The code to do this is shown below. It is the code we covered in class but I have moved things around slightly and commented on it so that I better understand what is going on.

```
# MAIN FUNCTION
def merge_sort(arr):

    # --------------
    # PART 1: DIVIDE
    # --------------

    # Base case for recursion - when the sub-array becomes a single
    # element, return it as an array (this is stored to the variable left
    # or right).
    if len(arr) <= 1:
        return arr

    # Get middle of array.
    mid = len(arr) // 2

    # Create two sub-arrays.
    left = arr[:mid]
    right = arr[mid:]

    # Recursive call on left sub-arrays then on the right sub-arrays once
    # left is stepped out of (returned). Values are stored in the
    # variables left and right on each recursive call - remember state is
    # maintained so as we step back out of each call, the earlier left
    # and right values are still there.
    left = merge_sort(left)
    right = merge_sort(right)

    # ----------------------
    # PART 2: MERGE (CONQUER)
    # ----------------------

    # Calls the merge function, detailed below.
    sorted_array = merge(left, right, arr.copy())

    return sorted_array
```

```
# HELPER FUNCTION
def merge(left, right, merged):
    # This function sorts the left and right sub-arrays and stores the
    # result in merged.

    left_cursor = 0
    right_cursor = 0
    merged_cursor = 0

    # This while loop looks at the first elements in the left and right
    # sub-arrays and places the smallest at the first position in merged
    # (i.e. at merged[0]). The element added to merged is removed from
    # future considerations (the cursor is incremented) and the element
    # not added is compared with the next element in the other sub-array.
    # While loop is stepped out of once all of the values in one of the
    # sub-arrays have been worked through.
    while left_cursor < len(left) and right_cursor < len(right):
        val_in_left = left[left_cursor]
        val_in_right = right[right_cursor]

        # If left value is smaller than or equal to the right value,
        # store the left value in merged and increment the left_cursor by
        # 1 so that this value is removed from consideration in future
        # iterations. Then proceed to the for loop below.
        if val_in_left <= val_in_right:
            merged[merged_cursor] = val_in_left  # overwrites value
            left_cursor += 1
            merged_cursor += 1

        # If right value is smaller, store this on the left and increment
        # cursors.
        else:
            merged[merged_cursor] = val_in_right  # overwrites value
            right_cursor += 1
            merged_cursor += 1

    # If there are any values in left that were not stored in merged,
    # iterate through them and append them to merged in the order that
    # they appear in left (these will already be in sorted order from
    # previous iteration).
    for left_cursor in range(left_cursor, len(left)):
        val_in_left = left[left_cursor]
        merged[merged_cursor] = val_in_left
        merged_cursor += 1

    # If there are any values in right that were not stored in merged,
    # iterate through them and append them to merged in the order that
    # they appear in right (these will already be in sorted order from
    # previous iteration).
    for right_cursor in range(right_cursor, len(right)):
        val_in_right = right[right_cursor]
        merged[merged_cursor] = val_in_right
        merged_cursor += 1

    return merged
```

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?

Key definitions

iterable – an object that is capable of returning its elements one at a time
iterator – an object that iterates over the elements of an iterable
generator function – a function that returns an iterator

Iterables and iterators

An iterable is an object that is capable of returning its elements one at a time, i.e. it can be iterated over (to iterate is to repeat a process again and again). Just because a Python object is composed of smaller individual elements does not mean that it can be iterated over. This functionality must be coded. For an object to have this functionality (i.e. be an iterable), it must belong to a class that has an `__iter__` method. This is the method returns an iterator object. The iterator object must have a `__next__` method that returns the next element of an iterable. Together, these two methods form the iterator protocol. These methods are called behind the scenes when a for loop is run. You can also call them directly or with the built-in functions `iter()` and `next()`:

```
>>> s = "CFG"

# create iterator with __iter__ method or iter function
>>> iterable = s.__iter__()
>>> iterable = iter(s)

# iterate over iterable with __next__ method or next function
>>> iterable.__next__()
'C'
>>> next(iterable)
'F'
>>> next(iterable)
'G'

# StopIteration is raised if next is called again after the last element
>>> next(iterable)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Several of the built-in Python data types (i.e. classes) have the iterator protocol coded into them, e.g. string, list, tuple, dictionary. You can also implement these methods within your own classes:

```
>>> class FourTimesTable:
...
... def __init__(self, max=12):
...     self.n = 1
...     self.max = max
...
... def __iter__(self):
...     return self
...
... def __next__(self):
...     if self.n > self.max:
...         raise StopIteration
...
...     result = 4 * self.n
...     self.n += 1
...     return result

# Example runs
>>> iterator = FourTimesTable(3)
>>> for val in iterator:
...     print(val)
4
8
12

# Each value can also be called with next function
>>> iterator = FourTimesTable()
>>> next(iterator)
4
>>> next(iterator)
8
>>> next(iterator)
12
```

Generator functions

As seen in the previous section, creating a custom iterator can be quite an involved process, requiring you to create a class and define the `__iter__` and `__next__` methods. Instead, you can use a generator function to create an iterator without having to go through the more lengthy process of designing and coding a class.

Normal functions use the `return` keyword to return something from the function and terminate it. A generator function uses the `yield` keyword. This ends the particular run of the function and saves its state. When the generator is called again, it recalls its saved state. It therefore knows where it was the last time it was called and returns the next value.

```
>>> def four_times_table(max=12):
...     n = 1
...     while (n-1) < max:
...         yield 4 * n
...         n += 1

# Example run
>>> for val in four_times_table(5):
...     print(val)
4
8
12
16
20
```

Alternatively, a generator object can be created with a generator expression. This is like a list comprehension but uses `()` instead of `[]`:

```
>>> four_times_table = (4 * n for n in range(1, 6))

# Example run
>>> for i in four_times_table:
...     print(i)
4
8
12
16
20
```

Use cases

The iterator protocol and generator functions can both be used to do the same thing. You would use the iterator protocol when you want to define the behaviour of an object at the class level. If you don't need to implement at the class level, a generator function or expression is easier to write, is terser and is easier to read.

Both are useful because they can be used to iterate over sequences without storing the sequence itself in memory. This means that very large sequences can be handled efficiently. For example, the built-in function `range()` is a generator function, enabling large ranges to be iterated over efficiently without storing each value in memory. Another example use case of this type would be reading in lines of text from a text file (or reading in rows from a csv file). One line/row can be yielded at a time by a generator function like this one:

```
def csv_reader(file_name):
    for row in open(file_name, "r"):
        yield row
```

As a sequence is not stored in memory, it means that infinitely large sequences can be handled, as would be generated, for example, by this function:

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

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

Decorators are a design pattern that can be applied to functions and classes in order to modify their behaviour without changing their underlying structure.

In Python, functions are first-class citizens. This means that they are treated like any other object – they can be assigned to variables, they can be used as arguments to other functions (like in the `reduce()` function covered question 7) and they can be returned from functions. Decorators take advantage of those last two points. They are functions that take a function as an argument and return another function, i.e. they wrap a function. In this way, they can be used to modify the behaviour of the function passed as an argument without explicitly changing that function itself. The original function can still be used where it is required in its original form, but the modified version is also available.

This simple example demonstrates how you can create a decorator function that extends the functionality of the original function without directly modifying the original. In this example, the original function `welcome()` prints a welcome message – "Welcome to Wales" – in English. The decorator function extends the function by invoking the original function and then printing the same thing but in Welsh – "Croeso i Gymru".

```
>>> def bilingual(func):
...     def inner():
...         func()
...         print("Croeso i Gymru")
...     return inner


>>> def welcome():
...     print("Welcome to Wales")
```

The original function is always callable:

```
>>> welcome()
Welcome to Wales
```

The `inner` function is what is returned by the decorator function. Note that it just returns the function it is not invoked. If we wanted anything to print out, we can assign the function to a variable and then invoke the variable:

```
>>> english_and_welsh = bilingual(welcome)
>>> english_and_welsh  # demonstrate that bilingual returns inner
<function bilingual.<locals>.inner at 0x10a7fd3a0>

>>> english_and_welsh()  # invoke bilingual
Welcome to Wales
Croeso i Gymru
```

Alternatively, we could have used the "@" symbol to decorate the `welcome()` function as it was being defined, so long as `bilingual()` was already defined:

```
>>> def bilingual(func):
...     def inner():
...         func()
...         print("Croeso i Gymru")
...     return inner


>>> @bilingual
>>> def welcome():
...     print("Welcome to Wales")

>>> welcome()
Welcome to Wales
Croeso i Gymru
```

Decorator ideas

- A decorator could be used to add strings of symbols (such as #, = or –) to certain print statements in a command line interface to separate different parts of a program and make it more readable.
- When we were doing the OOP homework, I was thinking I could have used a decorator to calculate the discounted price. The total price function would have been kept unchanged so we could always calculate the total pre-discount price. But I could have then decorated the total price function to make it calculate the total price after a discount. So decorators could be used in these types of situations where you need an original value but also a modified value like a discount or before and after VAT.
- They could be used to count the number of times a function is called.
- They could be used to time how long a function takes to run.
- They could be used to cache the results of a function.

Disadvantages

- Too many decorators can make it more difficult to maintain the code base.
- Caution is required with decorators as they can be harder to debug and make it more difficult to spot errors.