WEEK 4

Day 1

Inheritance

In Python, classes support inheritance, allowing a derived class to inherit attributes and methods from a base class.

 The syntax for defining a derived class is `class DerivedClassName(BaseClassName):`. Base classes are specified in parentheses.

 Attributes not found in the derived class are looked up in the base class, and this process is recursive if the base class itself is derived.

 Methods of base classes can be overridden in derived classes, and a derived class method can call the base class method using `BaseClassName.methodname(self, arguments)`.

Python provides `isinstance()` to check instance types and `issubclass()` to check class inheritance.

 Python also supports multiple inheritance, where a class can inherit from multiple base classes. The search for attributes follows a depth-first, left-to-right order.

The method resolution order dynamically adjusts to support cooperative calls to `super()`, allowing for more flexible and powerful multiple inheritance.

 Dynamic ordering is crucial to handle diamond relationships in multiple inheritance and ensures a reliable and extensible class hierarchy.

 In Python, there's no strict concept of "private" instance variables, but a convention is followed. Variables with a name prefixed by an underscore (e.g., `\_spam`) should be treated as non-public, an implementation detail subject to change without notice.

 For a limited form of name privacy, there's "name mangling." Identifiers of the form `\_\_spam` (two leading underscores, at most one trailing underscore) get replaced with `\_classname\_\_spam`, where `classname` is the current class name with leading underscores stripped. This helps avoid name clashes with subclasses.

Example:

```python

class Mapping:

    def \_\_init\_\_(self, iterable):

        self.items\_list = []

        self.\_\_update(iterable)

 def update(self, iterable):

        for item in iterable:

            self.items\_list.append(item)

 \_\_update = update   # private copy of the original update() method

 class MappingSubclass(Mapping):

    def update(self, keys, values):

        for item in zip(keys, values):

            self.items\_list.append(item)

```

 Here, even if `MappingSubclass` introduces a `\_\_update` identifier, it won't break `Mapping` because name mangling ensures it becomes `\_Mapping\_\_update` in `Mapping` and `\_MappingSubclass\_\_update` in `MappingSubclass`.

 However, note that name mangling is a convention, not a strict security measure. It's still possible to access or modify "private" variables, and in some cases, this can be intentionally used, such as in debugging.

 Keep in mind that when using `exec()`, `eval()`, or certain attribute-related functions like `getattr()`, `setattr()`, and `delattr()`, the mangling rules might not apply as expected.

 “Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. \_spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

 Since there is a valid use-case for class-private members (namely to avoid name clashes of names with namesdefined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form \_\_spam (at least two leading underscores, at most one trailing underscore) is textually replaced with \_classname\_\_spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

 Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

 Day2

 Iterators in Python allow looping over container objects using the `for` statement. The for statement implicitly calls `iter()` on the container, obtaining an iterator object with a `\_\_next\_\_()` method. When there are no more elements, `\_\_next\_\_()` raises `StopIteration` to signal the end of the loop. The `next()` function can also be used to manually retrieve elements from the iterator.

 You can implement iterator behavior in your classes by defining an `\_\_iter\_\_()` method that returns an object with a `\_\_next\_\_()` method. Generators provide a concise way to create iterators. They use the `yield` statement, allowing the function to resume where it left off when `next()` is

called. Generators automatically handle method creation, state saving, and termination, making them a compact and efficient tool for creating iterators.

The `sys` module in Python provides attributes for standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). The `sys.stderr.write()` function is useful for displaying warnings and error messages even when the standard output has been redirected. To terminate a script, `sys.exit()` is commonly used.

 For advanced string processing and pattern matching, the `re` module introduces regular expressions. It allows for complex matching and manipulation of strings. For instance, `re.findall(r'\bf[a-z]\*', 'which foot or hand fell fastest')` finds words starting with 'f' in a given sentence, and `re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')` removes repeated words.

 When only basic string operations are needed, using string methods is generally preferred for readability and simplicity. For example, `'tea for too'.replace('too', 'two')` replaces 'too' with 'two' in a given string.

 Python also offers modules for internet access and protocol processing. For retrieving data from URLs, `urllib.request` is used. The `smtplib` module facilitates sending emails. An example of using `urlopen` from `urllib.request` to fetch and display information from a URL is provided.

Day 3

The `datetime` module in Python provides classes for working with dates and times, supporting efficient member extraction for output formatting and manipulation. It also includes timezone-aware objects. The module allows easy construction, formatting, and arithmetic with dates.

 Data compression in Python is facilitated by modules such as `zlib`, `gzip`, `bz2`, `lzma`, `zipfile`, and `tarfile`. These modules support common archiving and compression formats.

 Performance measurement in Python is supported by the `timeit` module, which allows users to measure the execution time of code snippets. It provides a simple way to compare the performance of different approaches to the same problem.

For quality control, Python offers the `doctest` module for testing embedded tests in docstrings and the more comprehensive `unittest` module for maintaining a separate set of tests in a file. `doctest` allows tests to be included directly in the documentation, ensuring that the code remains consistent with its documentation.

In summary, Python's datetime module facilitates working with dates and times, data compression is supported by various modules, performance measurement can be done with timeit, and quality control is achieved through tools like doctest and unittest

Day 4

Logging, Managing packages with pip, and Floating point Arithmetic.

Logging:

Logging is a process of recording events, messages, or information about the execution of a program. It's a valuable tool for debugging and understanding the flow of a program. In Python, the logging module provides a flexible framework for emitting log messages from Python programs. You can use it to output messages to different places (console, files) at different levels (debug, info, warning, error, critical).

Managing Packages with pip:

 pip is the package installer for Python. Inside a virtual environment, you can use pip to install, upgrade, and uninstall packages. You can install, upgrade, and remove packages using a program called pip. By default pip will install packages from the Python Package Index, <<https://pypi.org>>. You can browse the Python Package Index by going to it in your web browser, or you can use pip’s limited search feature.

Virtual Environments:

 A virtual environment is an isolated Python environment where you can install packages and dependencies without affecting the global Python installation. This is useful to manage project-specific dependencies and versions. Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual

environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A’s environment.

 Creating Virtual Environments:

 To create a virtual environment, you can use the venv module (available in Python 3.3 and newer) or virtualenv (a third-party tool). Here's an example using venv. To create a virtual environment in Python, you can use the venv module, which is included in Python 3.3 and newer versions.

 Floating Point Arithmetic:

Floating-point arithmetic is a way to represent and perform arithmetic with real numbers (those that have a fractional part). In Python, floating-point numbers are represented as float data type. Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction 0.125 has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction 0.001 has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

PyPI · The Python Package Index

The Python Package Index (PyPI) is a repository of software for the Python programming language.