COMP1730/6730 Project Assignment

Introduction

This is 03.10.2022 version

In recent years, the assignment projects in COMP1730/6730 involved:

- modelling how the mainland of Australia (or its parts) are going to be consumed by rising sea level
- modelling the spread of fire across the ANU campus like it could have happened during the catastrophic January 2003 ACT bushfire
- trip planning and scheduling on the ACT public bus service
- modelling how the flooding waters will be distributed if such natural disaster occurs in the ANU campus
- modelling the dynamics of infection, illness, recovery and death caused by the COVID19 pandemic

These are all catastrophic scenarios (real or imaginary), having the potential of causing moral or psychological discomfort to students. Therefore, this time we are going to change the tune, we are going to choose something more cheerful,

something completely different.

We are going to write a software (a Python program) to study software (Python's Standard Library, StdLib). Python is an *interpreted* language which means that the original program (the source code) can be executed *directly* without being converted into a different format like it happens in *compiled* programming languages. (Not all code used in Python has such original form. Some components of the Python infrastructure exist in the compiled form, or as part of the Python interpreter program. But the majority of this infrastructure *is* Python code which can be read and analysed as text.)

Practical information

The assignment is due on **Friday the 28th of October at 9:00am, Canberra time** (the middle of semester week 12). Like all other due dates, this deadline is hard: **late submissions will NOT be accepted**.

For this assignment, you may work in groups of up to three students. **Working in larger groups (more than three students) is not allowed**. If there is an indication of four or more students working together, or sharing parts of solutions, all students involved will have to be investigated for possible plagiarism.

A group sign-up activity on Wattle is available until the **12th of October at 9:00am (Wednesday of semester week 10)**. If you intend to work in a group, you and your team mates should find a free group and add yourselves to it. (The group numbers have no meaning; we only care about which students are in the same group.)

Working in a group is not mandatory. If you want to do the assignment on your own, please add yourself to the "*I want to do the assignment on my own*" group, so that we can keep track. Remember that deadline extensions will only be given to individuals, not to groups. If you choose to work in a group, it is your responsibility to organise your work so that it cannot be held up by the unexpected absence of one group member.

Each student must submit two files:

1. Their code (a Python file). For students working in a group, it is required that all students in the group submit *identical* code files (with only difference being their name and ANU id).
2. An individual report, with answers to a set of questions. Details about the format of the report and the questions are in the section Questions for the individual report below.

Data and files provided

We provide you with no data files — all data will be "at your finger tips", to be extracted from the Python distribution which you use to perform standard programming activities. To be more precise: as a preliminary step, your program will compute the names of all modules and packages which are included in the Python Standard Library (*StdLib*). After obtaining the names, you will explore the contents of each *importable* module or package (we will specify the difference between these two terms below), and compute their mutual dependencies which will present a structural characteristic of Python software. This work will give you an opportunity to learn the resources of StdLib, which will be useful in your future Python programming endeavours. It will also help you to develop better understanding of the *namespace* concept, and difference and important connection between names and the objects they represent.

Structure of Python Standard Library

"Python comes with batteries included" — you might have heard this popular adage. The "batteries" here means the Standard Library of packages and modules, which are included in the canonical (*reference*) implementation called *CPython*, provided by the *Python Software Foundation*. A package or a module is a piece of code which has one of three forms:

- Pure Python code, a file (in case of a module) or a directory of files (in case of a package) with an extension (suffix) .py. Examples of such packages include collections, urllib, sqlite3 and others; examples of a single file modules are socket, csv, os and others. Some libraries have components, which exist in a compiled form.
- Compiled code in a form of shared libraries (dynamically loaded libraries, for Windows), with an extension .so (.ddl for Windows). Examples of such modules are math and array.
- Binary code included into the Python interpreter program itself (alongside with components of the __builtin__ module). Examples of such modules are itertools, time and sys.

Since there is no fundamental difference between Python code for a module and a package (a single file vs a directory of files), we will sometimes use a single term "package" or "module" meaning both of

them (in situation when this can cause a confusion, we will be more precise). The modules which exist in a compiled form (a shared/DDL library, or a part of the Python binary) can be called “packages”, too, since from the user’s perspective, it almost makes no difference what the actual nature of the package code is.

There is *programmatic way* to find out in which type of the above three a particular module is implemented, and we will be using it in our analysis.

Task 1: Collect names of the StdLib modules and packages

The list of all StdLib package names can be obtained programmatically. It depends on the Python version, like the list itself, because the contents of the StdLib changes from version to version — despite StdLib is quite stable, new packages sometimes get added, and (even more rarely) some packages can be dropped (after living in deprecated limbo for a few versions).

The most recent Python version 3.10 has a new feature which allows to obtain the package names list straight away: the StdLib package `sys` has an attribute called `stdlib_module_names` (an attribute is a package component like a function, but it represents an object, in this case, a `frozenset`, which is a collection of unique names). By evaluating `sys.stdlib_module_names` you will get a sequence of names representing the StdLib packages and modules.

If the Python *minor* version `x` (in the semantic notation `3.x`) is between 5 and 9, then the package `isort` can be used (`isort` is not in StdLib, but is included in Anaconda). Depending of the minor version `x`, the attribute which will give you the collection of StdLib package names is `isort.stdlibs.py3x.stdlib`. For example, for the version Python 3.8, the subpackage `isort.stdlibs.py38` has the attribute `stdlib` which is a collection (a set, to be precise) of StdLib packages.

The question now is how to make your program know which version it is. A *hint* — use the `sys.version_info` attribute to obtain the value of `x`, and *then* decide programmatically (ie, without the user intervention), which way to use — the one with `sys` module, or the one with `isort` — to obtain the list of StdLib package names.

Either way, using `sys.stdlib_module_names` for Python 3.10, or `isort.stdlibs.py3x` for Python 3.x, the list of names will contain *external* and *internal* packages. “External” here means the packages which are “exposed” to the external user. Apart from external packages, Python’s StdLib (like any standard API), has *internal* components, packages whose name starts with underscore. For example, some of you have seen an import statement `from __future__ import div`. `__future__` is one such “internal” package which normally is not used (and not meant to be used), yet it is available to help to deal with some special circumstances. (Unlike other languages, Python does not provide strict control for accessing internal API; Python relies on the *naming conventions*, when the names which follow a certain pattern, like having single or double underscore at the beginning, represent an internal function, class or module.) **The StdLib packages in our study will be only external packages.**

With this knowledge at hand, your task in the first part will be to write the function `get_stdlib_packages()` which determines the minor version of Python used to execute your program, and then chooses the right way to import the correct package (`sys` or `isort`) and obtain the set of StdLib package names. The function will not take any arguments, and it will return the names of external packages, those which **do not start with underscore _**.

One last twist: StdLib contains two external packages which are quite special — `this` and `antigravity`. They do not provide useful functions or classes, instead they generate an effect when imported (that is, they contain executable code, something we asked you not to include in homework code, and for a good reason). We advise you to explore what happens when you import them. The effect in one case will be some fun, another will display a collection of Python design principles (the so-called "Zen of Python", useful to read and think over). But when used in the follow up study, importing them will cause unnecessary noise. You should eliminate the possibility of such noise by filtering out the two names — `"this"` and `"antigravity"` — from the package name list, to make the function `get_stdlib_packages()` return the remaining names. You can convert it to a sorted list, or leave the original type set — it doesn't matter, it should be a sequence type object containing all available external StdLib package names (without the two spoilers).

As the last step, define the function `task1()` which takes no arguments and calls the function `get_stdlib_packages()` and uses its result to print the full version of the Python used to run the assignment code, the operating system name, and then first five and the last five StdLib package names. A possible output (we do not specify the exact format so you can choose your own) may look like this:

```
Python 3.9.13 on Linux arch 5.15.68-1-lts
StdLib contains 213 external modules and packages:
abc, aifc, argparse, array, ast ... zipapp, zipfile, zipimport, zlib, zoneinfo
```

(the exact number of packages may depend on Python's version).

Note It is possible that implementing a function requires some packages from StdLib, eg, in the case of `task1()` and `get_stdlib_packages()` packages `sys` and `platform` may be needed. Such "once-per-task" packages can be imported *locally* in the function body, which will keep them hidden from the global program namespace.

Task 2: Find what's in each module's namespace (by importing them)

The next task is to use the list of package names and determine which packages can *actually* be imported. Because, as it turns out, the StdLib contains packages which cannot be used on a particular software platform. For example, on Linux or Mac one cannot use packages which use Windows specific resources (hence, the failure to import packages like `winreg` or `winsound`). There may be different reasons for import failure, but such cases will be rare. How to find out if a package importable or not inside a program? Here, the advanced control flow mechanism, which is discussed in later lectures, can be useful. It involves the use of `try-except` statement to execute a *regular* code (in our case, the import

statement and the follow up use of the imported package), or execute the exception/error handling code, when the execution of the regular code fails. As Python statement, this looks as follows (again, we assume that you are familiar with this language feature):

```
try:
   <regular code execution>
except:
   <error handling code>
```

The regular code should be the one which imports a module, ie the statement like `import sys`. *However*, the `sys` here is not the string which represents the module name, it is the variable name of the module object itself (and Python knows it). Yet, in our collection of names, which was obtained in the Task 1, those names are strings. How to import a module by using its string name value instead the module name? To our help comes the StdLib module `importlib`, which provides the function `import_module` for doing exactly that. The execution of `importlib.import_module(module_name)` will return the module which can be used as if imported via the standard import statement. When the module cannot be imported, the function will raise an error (exception), that can be handled to prevent the program crash. In our case, we will only use error handling to determine than a particular module cannot be imported and used in further analysis.

In Task 2, you will write a function `get_real(package_names)` which will iterate through the sequence of names, and determine which ones represent module name that cannot be imported. It will exclude those names, and return a list of names of importable StdLib packages. The returned list must not be the same list object if such is passed as the argument. The returned names will be the names of packages to be explored further.

Note Every time your (successfully) import something, a module object in your program is created. You will do it multiple times (around 200), and almost every time the module will be used for a short task. It would be good if it is removed from the program. Python does such cleaning up itself by running a so-called “garbage collector”, but you may well do it yourself, by using the deletion operator `del` to delete the module after you established that it is importable, and therefore will be used again, but later. The statement `del a_module` will do just that — remove the module name from the program namespace, and free up the memory it used. Of course, the `del` operator can be used to delete any object by its name (`del` cannot delete a literal value). One module should probably be kept (since you import it explicitly) — `importlib`, because it will be needed again.

As the last part of this task, define the function `task2()` which takes no arguments and calls the function `get_real()` and uses its result to print the information about the packages which cannot be used on the execution platform (OS+Python version). Again, we do not specify an exact format, but this is one of the possible output:

```
These StdLib packages on Python-3.9.13/Linux arch 5.15.68-1-lts are not importable:
crypt, msilib, msvcrt, nt, winreg, winsound
```

Task 3: Calculate the linking/dependency between the StdLib modules

After we know what modules we can import, we can explore their contents and their mutual relationships.

A module is a namespace, where it keeps all its resources — useful functions, (rarely) useful literal values (like mathematical constants `e`, `pi` and `tau` in `math` module), custom defined types (created with `class` declaration), various exceptions and errors (which are types whose instances are created when a try-statement fails; you have been seeing them from the very beginning — remember `RobotError` from homework one?). After you imported a module, so its name exists in your program, the contents of its namespace can be obtained by using the builtin function `vars()` (there are other ways to do it, but this one should suffice). For example, the call `vars(math)` will return a dictionary, in which key-value pairs are name of package resource and the object this name represents. For example, `vars(math)['sin']` will return the function object which computes the value of *sine* function. Thus, by exploring the keys in the dictionary returned by `vars(a_module)` we can find the contents of the `a_module` namespace.

Some these names will be names of functions, or types, or modules from StdLib which the module `a_module` depends on. In Task 3, you will write a function `module_dependency(module_names, name)` which will take the list of names of importable StdLib packages (returned by the function in Task 2), and the name of a StdLib importable module (which should be among the names in `module_names`), and return the list of names on which the module `a_module` depends (all names in this list will be the StdLib module names). If `a_module` does not depend on any other module — we'll call it a *core module* — the returned list will be empty.

As a part of this task, define two functions which will use `module_dependency()`:

1. a function which will print names of five most dependent StdLib modules and the number of modules each of them depends on; one line of the output should look like this: `zipfile: 16`, the output should be sorted in descending order by the number of dependable modules (the number after colon in the example).
2. a function which returns the list of all core modules in StdLib

We leave it to you to find suitable names for these two functions.

Caveat Some StdLib modules/packages contain (in their namespace) names which are identical to the module name. For example, the *module* `time` contains the *function* `time`, or the *module* `socket` contains the custom *type* `socket`, and instances of this type can represent a socket connection between two network hosts. Such "duplicates" must not be confused and counted as modules.Your function `module_dependency()` should exclude such fictitious module dependency on itself.

As the last step, define the function task3() which takes no arguments and uses the function module_dependency() and the results from the two findings above to produce the following output (this format is not very strict, you can use a different one):

```
The following StdLib packages are most dependent:
zipfile:    16
subprocess: 15
pdb:        15
trace:      11
tarfile:    11
The 67 core packages are:
abc, argparse, atexit, audioop, binascii ... wsgiref, xml, xmlrpc, zlib, zoneinfo
```

(the exact numbers and names in the output may be different — do not rely on them as correctness check).

Task 4: Explore the code of Python's written modules

The next task will make us to go and look inside the module/package code itself. As we described in Structure of Python Standard Library section, there are three forms in which an StdLib resource exists in the "physical" space (as a file on the disk). We will only be interested in those modules, which are realised as Python code. For such Python based StdLib packages, their namespace provide a location of the corresponding files on the computer. Namely, in the case a single file, the module namespace will contain the __file__ attribute which gives the path to the corresponding file. If this file is Python code, its name will have suffix .py which can be used to read the file contents. If this file is a shared (DDL, for Windows) library, it will have .so (.ddl) suffix, and we will not attempt to read it. If the module is a part of builtin binary code, its names space will have neither __path__, nor __file__ attribute, and we also will not bother with it.

These properties of module/package namespace give us programmatic means to determine what code is used to implement a module, and if it's found to be Python code, we can find its location on the file system, and read it.

In Task 4, you will write a function explore_package(a_package), which will determine the type (Python code, shared library or binary code), and in the case of Python code, in either case of a single file or a directory, will read every Python file.

Note You will find the function os.walk in the os module useful to traverse the contents of the code directory in the case of Python-coded package. But remember that some files in such directory can be non-Python files, so you will have to check the suffix of every file which you traverse through.

The function explore_package() will return a tuple of values:

- the total number of lines in all Python files of a_package (0, if the package is not Python-coded)

- the total number of custom types a package defines; you will have to detect every class declaration (signified by the use of the reserved word `class`); the subtlety here is to avoid counting those `class` occurrences which are found *in the actual code inside docstrings or plain comments*.

As the last step, define the function `task4()` which takes no arguments and uses the function `explore_package()` and reports on the following findings:

- What are 5 largest packages in terms of the number of lines of code (LOC)?
- What are 5 smallest (Python coded) packages in terms of the LOC count?
- What are 5 largest packages in terms of the number of classes defined?
- What packages define no custom classes?

Format these outputs in a similar way to Tasks 1, 2 and 3.

Task 5: Cyclic module dependencies: Are there any in StdLib?

Software libraries and system sometimes contain *cyclic dependencies*, when a class or a function **A** depends on **B** which it turn depends on **C** … on **Z** which depends on **A**. Such dependencies are considered as design flaws, but not always (for example, in functional programming recursive dependencies is a standard construct).

Write a function `find_cycles()` which will compute all cyclic dependencies in the StdLib. Use this function in the function `task5()` to report on the findings, with the output looking like that:

```
The StdLib packages form a cycle of dependency:
1. package_A -> package_B -> package_C -> ...
2. package_D -> package_A -> package_Z -> ...
3. package_O -> package_Y -> package_B -> ...
```

The names `package_X` should be the StdLib package names found.

Task 6: Build the StdLib module connectivity graph (for COMP6730 only)

Using the `networkx` package to study graphs and networks and the plotting package `matplotlib`, create a visual representation of StdLib as a graph. The graph should show StdLib packages as nodes and their dependencies as links between the nodes. Save the (appropriately scaled) image in a `png` format, and include it into your report. The documentation for Networks library can be found online: NetworkX: Software for Complex Networks or elsewhere.

Finally

Assemble all the functions `task1()`–`task5()` (but not the function from Task 6, if you attempt it) into the function `analyse_stdlib()` to produce the output of findings in those tasks. You may conclude the program

with the “main” statement (as done in the template), but this is optional. The program project.py should be used as the template.

Questions for the individual report

A template for your report is provided here:

- assignment_report.txt

The template is a plain text file; write your answers where indicated in this file. **Do *not* convert it to doc, docx, pdf or any other format.**

The questions for you to answer in the report are:

- Report question 1: Write your name and ANU ID.
- Report question 2: If you are part of a group, write the names and ANU IDs of ALL members of this group. If you are doing the assignment on your own (not part of a group), just write “not part of a group”.
- Report question 3: Select a piece of code in your assignment solution that *you* have written, and explain:
  (a) What does this piece of code do?
  (b) How does it work?
  (c) What other possible ways did you consider to implement this functionality, and why did you choose the one you did?
  (d) Report your findings to the questions listed at the end of Task 4 description.
  For this question, you should choose a piece of code of reasonable size and complexity. If your code has an appropriate level of functional decomposition, then a single function is likely to be a suitable choice, but it can also be a part of a function. It should not be something trivial (like a single line, or a simple loop).
  For all parts of this question, it is important that your answers are at an appropriate level of detail. For part (a), describe the purpose of the code, including assumptions and restrictions. For parts (b) and (c), provide a high-level description of the algorithmic idea (and alternative ideas), not a line-by-line description of each statement.
  There is no hard limit on how short or how long your answer can be, but an answer that is short and clear is always better than one that is long and confusing, if both of them convey the same essential information. As a rough guideline, an appropriate answer may be about 100 – 300 words for each of parts (a) – (c).

Submission requirements

Every student must submit two files: Your assignment code (`project.py`) and your individual report (`assignment_report.txt`). An assignment submission link will be available on Wattle shortly after the assignment specification is released.

Restrictions on code

There are certain restrictions on your code:

- You should NOT use any global variables or code outside of functions, except for the test cases in the if __name__ == '__main__' section. You can add other test cases in the main section if you wish.
- You can import StdLib modules that you find useful. However, we will test your code using the CPython distribution of Python, so only modules available in it can be used (the only exception being the package isort, which is *not* a part of StdLib, but it is distributed in Anaconda). If in doubt about whether a module can be used, post a question to the Wattle discussion forum before you decide to use it.
- You are **NOT** allowed to use (in fact, you will not need it) any package/module which is not included into the StdLib. In particular, there will be no use of numpy, pandas and scipy packages and others which are included in the Anaconda distribution. **An exception is the package isort needed for extracting the list of StdLib package names (as expalined above), and the packages networkx and visualisation packages like matplotlib which can be used in Q6 for COMP6730 students.** The networkx documentation can be found on https://networkx.org/.

It is very important that you follow these restrictions. If you do not, we may not be able to test your code, in which case you can not gain any marks for code functionality.

Assumptions

We will test your code using standard Python distribution. It may be installed on a computer which runs different (from yours) operating system; it may be a different version of Python programming language (but it will be reasonably modern version of Python 3. Therefore, the results which will be produced from a correctly written program may have slight differences from those which will be obtained by you (using a different machine and/or different version of Python). But, the differences will be minute, because the Standard Library is very stable, and the OS specific differences are small and well known.

Referencing

In the course of solving the assignment problem, you will probably want to make use of all sources of knowledge available: the course materials, text books, on-line python documentation, and other help that you can find on-line. This is all allowed. However, keep in mind that:

- If you find a piece of code, or an algorithmic idea that you implement, somewhere on-line, or in a book or other material, you must reference it. Include the URL where you found it, the title of the book, or whatever is needed for us to find the source, and explain how you have used it in an appropriate place in your code (docstring or comment).
- Although you can often find helpful information in on-line forums (like *StackExchange*, for example), *you may not use them to ask questions that are specific to the assignment*. Asking someone else to solve an assignment problem for you, whether it is in a public forum or in private, is a form of plagiarism, and if we find any indication that this may have occurred, we will be forced to investigate.

- If you have any doubt about if a question is ok to ask or not, you can always post your question to the Wattle discussion forum. We will answer it at a level of detail that is appropriate.

Remember that:

- Working in groups of more than three students is **not allowed**. While you may develop your solution (python code) in a group of up to three students, you may not share your solution, or parts of your solution, with, or receive parts of a solution from, anyone outside this group.
- The individual report **must be done by you yourself**. The other members of your group may not assist you with it.

Marking criteria

The code component accounts for 95% of the total assignment marks, and your individual report for the remaining 5%. For the questions of the code component, the breakdown is as follows: (**this will be edited to conform to the problems in this assignment**)

- Q1: 15%
- Q2: 20%
- Q3: 20%
- Q4: 20%
- Q5: 20%
- Q6: 20% (COMP6730 total mark will be scaled down to 95%)

Your code will be marked on two criteria: Functionality and code quality. The division of marks between them is 60% for functionality and 40% for code quality. We will also consider the efficiency of your code. Efficiency is part of both functionality and code quality.

Functionality encompasses code running without error on input files that follow the same format as the example files provided, and produces an output that is easy to understand and correct. Examples of increasing levels of functionality are:

- Code runs without runtime error, and in a reasonable amount of time, on the provided example data files.
- The output is understandable, and includes the information that is asked for in the question.
- The output is correct for a standard version of Python and Operating System.
- Code runs without runtime error, and in a reasonable amount of time.

We do not specify a precise limit for what is “a reasonable amount of time”, but as a guide, your code should process the example data files in no more than a minute or two. If it takes much longer, then we will not be able to test your code fully, which is practically the same as it not running at all.

Code quality includes the aspects that we have discussed in lectures and homework:

- Good code documentation. This includes appropriate use of docstrings and comments.

- Remember that for some questions you will have to make some assumptions, and decisions how to calculate your estimate. You should describe these in the code, using docstrings and comments as appropriate.
- Good naming. This includes variable and function names.
- Good code organisation, including appropriate use of functional decomposition. Remember that even though the assignment template has only one function that you must implement, you can define other functions and use them in your solution.
- Efficiency. This includes things like considering the complexity of the operations that you use, and avoiding unnecessarily slow methods of doing things. We do not require that every part is implemented in an optimally efficient way, but code that has many or large inefficiencies is considered to have lower quality.