Lecture Agenda and Notes Draft:
Code Quality for Software

1. Code quality aspects [5 min]

There are multiple aspects of source code quality. Some of the aspects are easy to ensure while the other are very difficult to guarantee.

One way of trying to define code quality is to look at one end of the spectrum: high-quality code. Hopefully, you can agree on the following high-quality code identifiers:

- It does what it is supposed to do.
- It does not contain defects or problems.
- It is easy to read, maintain, and extend.

This neat article on code quality explains a bit on why these basic code quality formulations make sense.

This also explains the commonly accepted paradigm in software engineering:

> "*Make It Work, Make It Right, Make It Fast*"

This formulation of this statement has been attributed to Kent Beck; it has existed as part of the UnixWay for a long time ("the strategy is definitely: first make it work, then make it right, and, finally, make it fast.")

This means roughly:

- Get at least some of the stuff to work so you are getting feedback (i.e. get your first test case to pass)
- From there, get everything to work so you have completed a chunk of functionality
- *Optional: make it go faster, but only if you need to.*

Today we focus on the following three important aspects of code quality: existence of bugs in a program, program performance, and source code quality.

2. Debugging in Python [30 min]

Bugs are errors in code, and they are ubiquitous reminders that machines, by their very nature, do exactly what we tell them to do. Therefore, bugs are typically imperfections in syntax and logic *introduced by humans*. However careful we are, bugs *will be* introduced while we are developing code. They begin to be introduced as soon as we start writing a piece of code. For this reason, we must be vigilant, and we must be prepared to fix them when they arise.

This class will prepare you to recognize, diagnose, and fix bugs using various approaches and tools for "debugging" your code. It will do so by introducing:

1. When, how and how bugs are encountered.
2. Methods of diagnosing bugs.
3. Interactive debugging, for diagnosing bugs quickly and systematically.

4. Profiling tools to quickly identify memory management issues.
5. Syntax analysis tools to catch style inconsistencies and typos.

Of these, the most time will be spent on using the **pdb** interactive debugger in Python, since it is essential for debugging issues whose source is not obvious from tests or simple print statements. First, however, we will discuss the instigating event itself: encountering a bug.

Encountering a Bug

Bug may take the form of incorrect syntax, imperfect logic, an infinite loop, poor memory management, failure to initialize a variable, user error, or myriad other human mistakes. It could be in form of:

- An unexpected error while compiling code
- An unexpected error message while running the code
- An unhandled exception from a linked library
- An incorrect or invalid result
- An indefinite pause or hang-up
- A full computer crash
- A segmentation fault
- and many others

Print Statements

`print` statements are every developer's first debugger. Printing is typically a check that asks one or both of these questions:

- Is the bug happening before a certain line?
- What is the status of some variable at that point?

In the following example, something about the code is causing it to "hang". Its likely you can determine the cause of this problem by fastly looking at it, however, in the case you can't, a `print()` statement can be inserted where code is suspected to be hanging:

```
def mean(nums):
    bot = len(nums)
    it=0
    top=0
    while it < len(nums):
            top += nums[it]
    return float(top) / float(bot)
if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10]
    mean(a_list)
```

```
def mean(nums):
    bot = len(nums)
    it=0
    top=0
    print("Still Running at line 5") (1)
    while it < len(nums):
            top += nums[it]
            print(top) (2)
    return float(top) / float(bot)
if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10]
    mean(a_list)
```

- The print() is added to determine where the error is happening. This statement executed successfully as we caught the error before the troublemaking line.

- This one is added to determine what is happening to the variables during the loop. It will be printed infinitely.
- Can you tell what is wrong in the code using this information?

The infinite loop can be easily fixed in the code:

```
def mean(nums):
    top = sum(nums) (1)
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10]
     print(mean(a_list))
```

- (1) Rather than looping needlessly, the sum() function can be applied to the list.

Thus print() can provide helpful information for pinpointing an error – but know that is not the best practice for effective computing, because, in a large code base, it will take more than a few print() to determine the exact line at which the error occurred. A more scalable solution is presented in the next section.

Interactive Debugging

Rather than littering one's code base with *print()* statements, interactive debuggers allow you to pause during execution and jump into the code at a certain line of execution.

Interactive debugging tools give the following functionality:

- Query the values of variables
- Alter the values of variables
- Call functions
- Do minor calculations
- Step line by line through the call stack

Interactive debugger in Python standard library *(pdb)*

Python's pdb provides an interactive prompt where code execution can be paused with the trace and subsequent breakpoints.

Let's take the example from the previous part, but add one more element to list:

```
def mean(nums):
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10, “one hundred”]
     print(mean(a_list))
```

running this returns an error:

```
Traceback (most recent call last):
  File "a_list-mean.py", line 8, in <module>
    mean(a_list)
  File "a_list-mean.py", line 2, in mean
    top = sum(nums)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

It seems that error has to do with the types of the values in the list. Let's import `pdb` and check if changing the non-`int` list values to a number will resolve the error. Additionally, we must provide starting points where we would like to "jump into" the execution and set a `trace`.

Setting the trace

Rather than inserting a new print statement in a new line every time new information about execution is uncovered, you can set a trace ping at the line where you would like to enter the program interactively in the debugger, by doing:

```
pdb.set_trace()
```

for example,

```
def mean(nums):
    bot = len(nums)
    return float(top) / float(bot)
if __name__ == "__main__":
    pdb.set_trace() (1)
    a_list = [1, 2, 3, 4, 5, 6, 10, “one hundred”]
     print(mean(a_list))
```

Now when the script is run, the Python debugger starts up and drops us into the code execution at that line.

```
   a_list-mean.py(10)<module>()
-> a_list = [1, 2, 3, 4, 5, 6, 10, 'one hundred']
(Pdb)
```

What to do next?

Type and enter `help` to get a table of commands:

```
(Pdb) help

Documented commands (type help <topic>):
========================================
EOF     c          d        h         list      q        rv       undisplay
a       cl         debug    help      ll        quit     s        unt
alias   clear      disable  ignore    longlist  r        source   until
args    commands   display  interact  n         restart  step     up
b       condition  down     j         next      return   tbreak   w
break   cont       enable   jump      p         retval   u        whatis
bt      continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb)
```

Type and enter `next (n)` to continue execution until the next line and stay within the current function, i.e. not stop in a foreign function if one is called. Think of next as "staying local" or "step over".

Type and enter `step (s)` to execute the current line and stop in a foreign function if one is called. Think of step as "step into". If execution is stopped in another function, s will print --Call--.

```
a_list-mean.py(11)<module>()

-> print(mean(a_list))
```

Simply typing the name of the variable will return it state

```
(Pdb) a_list

[1, 2, 3, 4, 5, 6, 10, 'one hundred']
```

Now it's clear that the variable that we set is suspect. The error we received involved a type mismatch during the summation step. During pdb we can change the state of the variable and run functions:

```
(Pdb) a_list[-1] = 100

(Pdb) a_list

[1, 2, 3, 4, 5, 6, 10, 100]

[1, 2, 3, 4, 5, 6, 10, 100]

(Pdb) mean(a_list)

16.375
```

With command `continue (c)` we can go through the rest of the code until the program ends. The final element of list is no longer a string (it has been set to the integer 100), the execution should succeed. Or it will stop on first break, set earlier, we will go through it later.

Actual file can be edited to capture this bug fix.

During executing `step`, being inside function you can call `args (a)` to print all arguments function

Breakpoints

What if a variable should be checked at many points in the execution -- perhaps every time a loop is executed, every time a certain function is entered, or right before as well as right after the variable should change its value.

In pdb, we can set a breakpoint using the `break (b)`. We set it in a certain line in the code by using the line number of the place in the code or the name of the function to flag. You can use `list (b)` to see context to easily navigate which line to set break:

```
b(reak) ([file:]lineno | function)[, condition]
```

set break at line :4 and :6 and continue till entering the first breakpoint.

Modifying program execution flow with pdb

The python debugger lets you change the flow of your program at runtime with the `jump` command. This lets you skip forward to prevent some code from running, or can let you go backwards to run the code again. We will be working with a small program that creates a list of the letters contained in the string 'dummy'.

```
def dummy_print():

    dummy_list = []
    dummy = 'dummy'
    for letter in dummy:
        dummy_list.append(letter)
        print(dummy_list)

if __name__ == "__main__":
        dummy_print()
```

Invoking pdb with pdb `python -m pdb dummy_letters.py`

Various Commands for pdb

1) list content:

```
(Pdb) l
  1  ->     def dummy_print():
  2           dummy_list = []
  3           dummy = 'dummy'
  4           for letter in dummy:
  5               dummy_list.append(letter)
  6               print(dummy_list)
  7
  8       if __name__ == '__main__':
  9           dummy_print()
[EOF]
```

2) set break to

```
(Pdb) b 5
Breakpoint 1 at dummy_list.py:5
```

3) `c` to continue till first break

```
(Pdb) list
  1   def dummy_print():
  2           dummy_list = []
  3           dummy = 'dummy'
  4           for letter in dummy:
  5 B->           dummy_list.append(letter)
  6               print(dummy_list)
  7
  8       if __name__ == '__main__':
  9           dummy_print()
[EOF]
```

4) `pp` - stands for pretty prints, you can print values passed to variable. Lets print `pp letter` var:

```
(Pdb) pp letter
'd'
```

5) `c` – continue through loop to line 5. till print command

```
(Pdb) c
['d']
```

6) `jump 6` - to skip to line 6.

```
> dummy_list.py(6)dummy_print()
-> print(dummy_list)
```

7) `disable 1` - disables breakpoints

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/vyacheslav/dummy_list.py:5
      breakpoint already hit 2 times
(Pdb) disable 1
Disabled breakpoint 1 at /Users/vyacheslav/dummy_list.py:5
```

8) `c` - continue

```
(Pdb) c
['d']
['d', 'm']
['d', 'm', 'm']
['d', 'm', 'm', 'y']
```

As soon as we disabled the brakepoint to proceed with execution as usual with the `continue` command, so 'u' never appended to `dummy_list()`

3. Profiling in Python [15 min]

Profilers are used to sketch a profile of the time spent in each part of the execution stack. Profiling goes hand in hand with the debugging process. When there are suspected memory errors, profiling is the same as debugging.

In Python, `cProfile` is a common way to profile a piece of code.

```
python -m cProfile -o output.prof dummy_letters.py
```

This creates a profile file `output.prof` in a binary format which must be read by an interpreter of such files. To view the profile file, we can use `pstats` package:

```
In [1]: import pstats
In [2]: p = pstats.Stats('output.prof')
```

In [3]: p.print_stats()

```
Fri Oct  8 13:36:00 2021    output.prof

           14 function calls in 0.000 seconds

    Random listing order was used

    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
         5    0.000    0.000    0.000    0.000 {method 'append' of 'list'
objects}
         1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
         5    0.000    0.000    0.000    0.000 {built-in method builtins.print}
         1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
         1    0.000    0.000    0.000    0.000 dummy_list.py:1(dummy_print)
         1    0.000    0.000    0.000    0.000 dummy_list.py:1(<module>)
```

`print_stats` function prints the number of calls (`ncalls`), the total time spent in each function (`tottime`), the time spent each time that function was called (`percall`), the cumulative time elapsed in the program, and the place in the file where the call occurs.

One can also use the profiler with the

```
# Sort output by Cumulative time
if __name__ == '__main__':
    import cProfile, pstats
    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats('cumtime')
    stats.print_stats()
```

The view is not very informative for simple programs like ours, but does that take longer to run. The zeros in this example indicate that the time per function was never higher than 0.0009sec.

A different format can be set with method `f8` if pstats.

```
from pstats import SortKey

p.sort_stats(SortKey.CUMULATIVE).print_stats(10).strip_dirs()
```

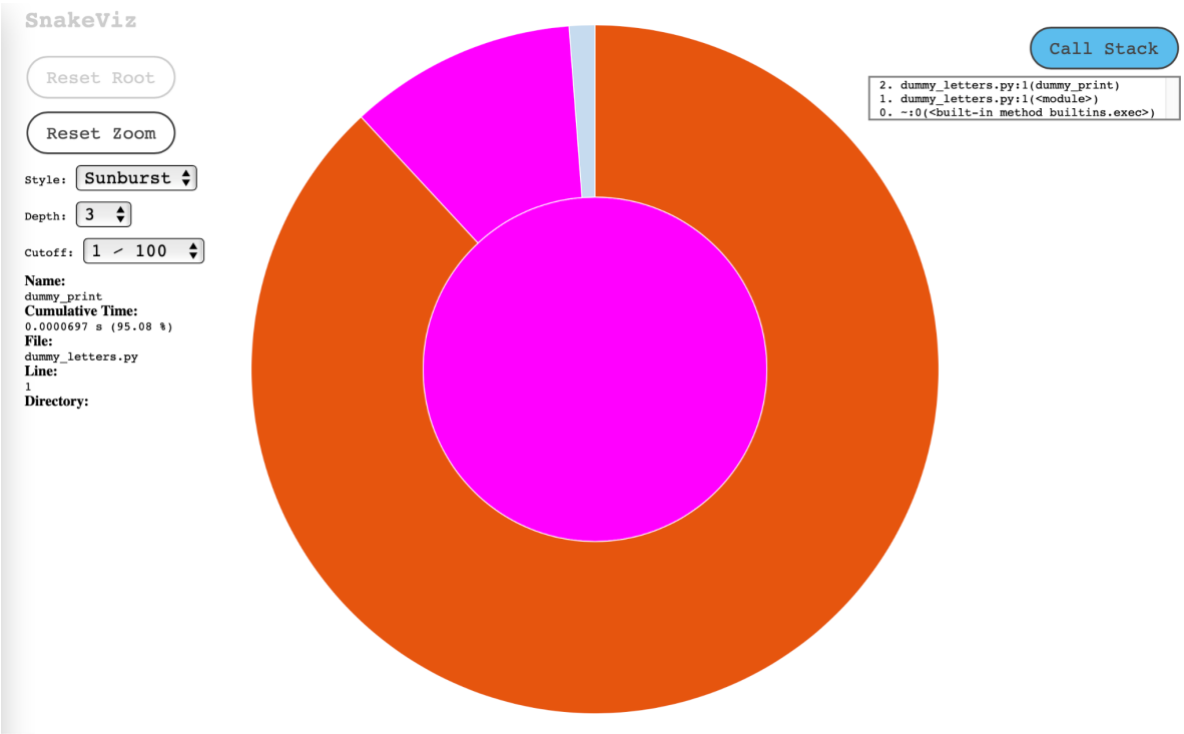
Change the output format of seconds

```
def f8(x):

     ret = "%8.3f" % x

     if ret != '   0.000':

         return ret

     return "%6dµs" % (x * 10000000)


pstats.f8 = f8
```

Viewing the Profile Graphically

SnakeViz package will cause a web browser to open and will provide interactive infographics of the data in output.prof.

With SnakeViz, the execution of the code can be browsed on a function-by-function basis. The time spend in each function is rendered in radial blocks. The central circle represents the top of the call stack - that is, the function from which all other functions are called. In our case, that is main body of the module in the final four lines of the file.



4. Linting and source quality [20 min]

Linting removes “lint” from source code. **Linting** is the process of checking the source code for Programmatic as well as Stylistic errors. This is most helpful in identifying some common and uncommon mistakes that are made during coding.

> *“The term **lint** was derived from the name of the **undesirable bits of fiber and fluff** found in sheep's wool.”*
>
> *Lint was the name of a program that would go through your C code and identify problems before you compiled, linked, and ran it.*

It can be helpful at each of these stages of the programming process. Linting catches unnecessary imports, unused variables, potential typos, inconsistent style and other similar issues. Linting is the process of running a program that will analyse code for potential errors.

> *lint was the name originally given to a particular program that flagged some suspicious and non-portable constructs (likely to be bugs) in C language source code. The term is now applied generically to tools that flag suspicious usage in software written in any computer language.*

Why is linting important? Linting is important to reduce errors and improve the overall quality of your code. Using lint tools can help you accelerate development and reduce costs by finding errors earlier.

The usage of linters has also helped many developers to write better code for not compiled programming languages. As there is not compiling time errors, finding typos, syntax errors, uses of undeclared variables, calls to undefined or deprecated functions, for instance, helping developers to fix it faster and reduce bugs before execution.

Here are the aspects of code quality that the modern linters can support:

- **Static analysis of source code.** Static analysis means that automated software runs through your code source without executing it. It statically checks for potential bugs, memory leaks, and any other check that may be useful.
  Static analysis involves computing software/code metrics such as number of lines of code (LOC), Cyclomatic Complexity, Maintainability Index, Halstead Metrics, etc.
- **Making the code adhere to a particular writing standard (standardizing).** For this purpose, in Python a few guides have been proposed to improve the quality of code (see below). Pylint, flakes8
  - PEP 8 — the Style Guide for Python Code https://pep8.org
  - Google Python Style Guide: https://google.github.io/styleguide/pyguide.html
- Finding and fixing security issues.
- Finding and fixing performance issues.

Automatic Tools for Linting

Linting can be achieved with the many many tools, some of them focusing more on one aspect but not the other. Below is a brief enumeration of the relevant linting tools along with their explanation. More examples are in this blog post.

- PyType is relevant for a dynamically typed language that Python is and allows checks and infers types for your Python code - without requiring type annotations.
- Radon, a static code analysis tool.
- Black, for code reformatting.
- PyFlakes Pyflakes analyzes programs and detects various errors. The information is more than just cosmetic since importing packages takes time and occupies computer memory, reducing unused imports can speed up your code.

```
>pyflakes  dummy_letters.py
dummy_letters.py:1:1 'sys' imported but unused
dummy_letters.py:2:1 'os' imported but unused
```

- pycodestyle is a tool to check your Python code against some of the style conventions in PEP 8. It will analyze the Python code that you have provided and will respond with a line-by-line listing os stylistic incompatibilities with the pep8 standard:

```
>pycodestyle dummy_letters.py
dummy_letters.py:4:1: E302 expected 2 blank lines, found 1
dummy_letters.py:4:19: W291 trailing whitespace
dummy_letters.py:10:1: E101 indentation contains mixed spaces and tabs
dummy_letters.py:10:1: W191 indentation contains tabs
dummy_letters.py:10:1: W293 blank line contains whitespace
dummy_letters.py:11:1: E305 expected 2 blank lines after class or function
definition, found 1
```

```
dummy_letters.py:12:1: E101 indentation contains mixed spaces and tabs
```

- PyLint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells. It can also look for certain type errors, it can recommend suggestions about how particular blocks can be refactored and can offer you details about the code's complexity.