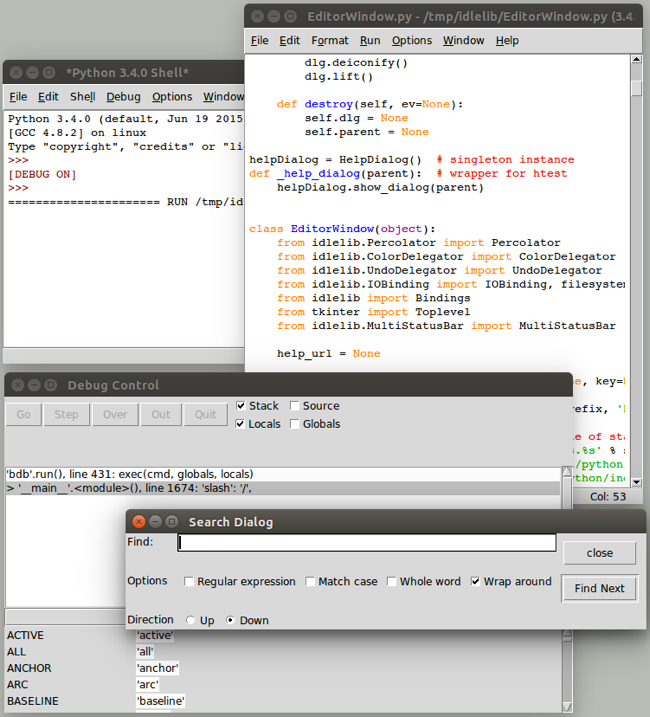
**PYTHON PROGRAMMING**

**ASSIGNMENT WEEK 1**

1.Case study on Python IDLE.

IDLE(Integrated DeveLopment Environment) is the standard Python development environment that is bundled with every Python release. It consists of an interactive Python shell, editors with syntax highlighting, a debugger, etc. Its user interface is written in Tkinter.

**  
Overview of IDLE user interface (on Linux).**

IDLE was never intended to be a replacement for more full-featured development environments. Because it is relatively simple and bundled with Python, it is popular for those learning (and teaching) the language.

Originally created by Python BDFL Guido van Rossum in 1998, IDLE has been incrementally added to over the years by multiple other developers. But with limited development effort spent on it, it was showing its age, especially on platforms (e.g., macOS) infrequently used by those improving IDLE.

A [Python Central comparison of IDE's](https://pythoncentral.io/comparison-of-python-ides-development/) described IDLE this way:

*All those features are, in fact, present, but they do not really make an IDE. In fact, while IDLE offers some of the features you expect from an IDE, it does so without even being a satisfactory text editor. The interface is buggy and fails to take into account how Python works, especially in the interactive shell, the auto-completion is useless outside the standard library, and the editing functionality is so limited that no serious Python programmer — heck, no serious typist — could use it full-time.  
   
If you use an IDE, it should not be IDLE.*

With its buggy and dated user interface, IDLE was [at risk](https://mail.python.org/pipermail/idle-dev/2015-July/003534.html) of being removed from the Python distribution altogether. Yet, because it is simple and bundled, many people, particularly those teaching Python, were eager to see IDLE leap forward.

IDLE was obviously a great candidate to be modernized, using newer Tk features like the themed widgets to help spur some redesign. But it was about more than just swapping widgets. Many improvements could be made just by changing how the "classic" widgets were being used to better reflect a more modern design aesthetic.

2.Discuss on PEP - Index of all Python Enhancement Proposals.

**PEP** is an abbreviation for **Python Enhancement Proposal**. A PEP is a design document that informs the Python community or describes a new feature for Python, its processes, or its environment. The PEP should provide a brief technical description of the feature as well as its reasoning. PEPs are intended to be the primary mechanisms for proposing important new features, gathering community input on a problem, and documenting Python design decisions.

The PEP author is responsible for building consensus within the community and documenting opposing (dissenting) opinions.

Because the PEPs are stored as text files in a versioned repository, their revision history serves as the feature proposal's historical record. This historical record can be accessed using standard git commands to get previous versions, and it can also be browsed on GitHub.

Types of PEP

PEP is classified into three types −

* A **Standards Track** PEP defines a new Python feature or implementation. It may also specify an interoperability standard that will be supported outside of the standard library for current Python versions until a future PEP adds standard library support.
* An **Informational** PEP addresses a Python design issue or offers general guidance or information to the Python community without proposing a new feature. Users and implementers are allowed to ignore or follow Informational PEPs because they do not necessarily represent a Python community consensus or recommendation.
* A **Process PEP** describes a Python-related process or proposes a change to (or event in) a process. Process PEPs are similar to Standards Track PEPs in that they apply to areas other than the Python language. They may suggest an implementation, but not to the Python codebase; they frequently require community consensus; thus, unlike Informational PEPs, they are more than recommendations that users are not free to disregard/ignore them. Procedures, guidelines, changes to the decision-making process, and modifications to the tools or environment used in Python development are all examples. Any meta-PEP is a Process PEP as well.

Workflow of PEP

Steering Council of Python

This PEP makes frequent references to the "Steering Council" or "Council." This refers to the current members of the elected Steering Council mentioned in PEP 13, who serve as the final decision-makers on whether or not PEPs are accepted or rejected.

Python’s Core Developers

This PEP makes many references to "core developers." This relates to the Python core team members who are currently active, as defined in PEP 13.

BDFL of Python

Previous versions of this PEP referred to PEP decision makers as "BDFL-Delegate." This was a historical reference to Python's old governance model, in which all design authority ultimately originated from Guido van Rossum, the Python computer language's original developer. The design authority of the Steering Council, on the other hand, is derived from their election by the currently active core developers. PEP-Delegate is now utilised instead of BDFL-Delegate.

PEP Editors

PEP editors are those in responsible for managing the administrative and editorial parts of the PEP workflow (e.g. assigning PEP numbers and changing their status). For more information, see PEP Editor Responsibilities & Workflow.

PEP editorship is by invitation of the present editors, who can be reached on GitHub by mentioning @python/pep-editors. The PEP workflow can be completed entirely through the GitHub PEP repository problems and pull requests.

Beginning with an idea for Python

The PEP process begins with a new Python idea. A single PEP should contain a single important proposal or new concept; the more concentrated the PEP, the more successful it is likely to be. The majority of improvements and bug fixes do not require a PEP and can be reported straight to the Python issue tracker. The PEP editors reserve the discretion to reject PEP proposals that appear to be too broad or unfocused. When in doubt, divide your PEP into multiple concentrated ones.

Each PEP requires a champion, who writes the PEP in the manner and format defined below, moderates discussions in the appropriate forums, and strives to develop community consensus around the concept. The PEP champion (also known as the Author) should first determine whether the idea is PEP-able. Unless a more specialized arena, such as Typing-SIG for static typing or the Packaging category of the Python Discourse, posting to the Ideas category of the Python Discourse is usually the best way to go about this.

The purpose of publicly evaluating a concept before going as far as writing a PEP is to save the potential author time. Many ideas for modifying Python have been proposed and rejected for various reasons. Asking the Python community first if a proposal is original helps to avoid wasting time on something that is almost certain to be rejected based on previous conversations (searching the internet does not always do the trick). It also aids in ensuring that the idea is applicable to the entire community rather than just the author. Just because an idea seems excellent to the creator does not guarantee that it will work for the majority of people in the majority of regions where Python is used.

After the champion has asked the Python community whether an idea has a possibility of being accepted, a drafted PEP should be given to the above-mentioned venue. This allows the author to fill out the draft PEP so that it is correctly formatted, of high quality, and addresses any early concerns about the proposal.