

Introduction to Python 2024

Lyda Hill Department of Bioinformatics

Course Overview

Day 1

- Opening Remarks
- Practical Tools for Running Python Software
- Data Structures
- Control Statements
- Functions
- Dependency Management and Environment Requirements

Day 2

- Introduction to the Python Ecosystem of Libraries
- Reading and Writing Data
- Data Visualization
- Introduction to numpy
- Basics of Machine Learning

Why Python?

Why Python?

Modern Software and Scientific Programming.

- **Python's Popularity:**

- One of the most widely used programming languages globally.
- Preferred in academia, industry, and research for its simplicity and versatility.

- **Ease of Learning:**

- Intuitive syntax that resembles human language.
- Extensive documentation and active community support.
- Ideal for beginners but powerful enough for experts.

Why Python?

Modern Software and Scientific Programming.

- **Broad Applicability:**

- Used across multiple domains: web development, data science, automation, scientific research, and more.
- Extensive standard library and third-party packages available on PyPI (Python Package Index).

- **Rapid Development:**

- Enables quick prototyping and iterative development.
- Reduces the time from concept to implementation, especially in research settings.

Why Python?

Modern Software and Scientific Programming.

- **Open Source and Free:**

- Large ecosystem with a vast number of open-source libraries and tools.
- Access to data science, machine learning, and artificial intelligence tools.
- Accessible and affordable for educational and research purposes.

- **Scientific Computing & Research:**

- Essential for data analysis, simulation, and modeling in scientific research.
- Libraries like NumPy, SciPy, Pandas, and Matplotlib are standard in the biomedical field.

Why are you interested in Python?

Activity

Install Anaconda

www.anaconda.com/download/

Practical Tools for Running Python Software

Executing Python Software

Practical Tools for Running Python Software

Command Line Execution

Using `python script.py` or running the Python interpreter directly from the terminal.

- **Step 1: Open the Command Line.**

- Windows: Open Command Prompt (search for `cmd` or `Command Prompt`).
- macOS/Linux: Open Terminal.

- **Step 2: Navigate to your Working Directory.**

Use the `cd` command to navigate to the folder where you want to create and execute your Python script.

```
cd path/to/your/folder
```

Practical Tools for Running Python Software

Command Line Execution

- **Step 3: Create a Simple Python Script.**

Use a text editor to create a new Python script called `hello.py` with the following content.

```
# hello.py
print("Hello, World!")
```

- **Step 4: Execute the Python Script.**

Run the script using the following command:

```
python hello.py
```

Practical Tools for Running Python Software

Command Line Execution

- **Advantages:**

- Simple and quick for running standalone scripts.
- Great for automation and batch processing.
- Efficient for executing complete programs.

- **Disadvantages:**

- Limited debugging capabilities.
- No interactivity once the script is running.
- Less suitable for exploratory analysis or iterative development.

Practical Tools for Running Python Software

Interactive Mode

Entering the Python interpreter by typing `python` in the terminal and running code line-by-line interactively.

- **Step 1: Open the Command Line.**

- Windows: Open Command Prompt (search for `cmd` or `Command Prompt`).
- macOS/Linux: Open Terminal.

- **Step 2: Start the Python Interpreter.**

Enter the following command to start the interactive mode:

```
python
```

- Once in the interactive mode, your terminal will output information on the `Python` installation.

```
Last login: Thu Oct 10 08:44:41 on ttys000
(base) S155475@SW567797 slides % python
Python 3.9.12 (main, Apr 5 2022, 01:52:34)
```

Practical Tools for Running Python Software

Interactive Mode

Entering the Python interpreter by typing `python` in the terminal and running code line-by-line interactively.

- **Step 3: Test a Simple Python Command.**

Once the interpreter starts, you can run Python code line-by-line. For example:

```
>>> print("Hello, World!")
```

- **Step 4: Define a Simple Variable and Perform a Calculation.**

Enter commands directly into the interpreter:

```
>>> x = 5
>>> y = 10
>>> result = x + y
>>> print(result)
```

- **Step 5: Exit the Python Interpreter.**

Practical Tools for Running Python Software

Interactive Mode

- **Advantages:**

- Immediate feedback; great for testing small snippets of code.
- Excellent for learning and experimentation.
- No need to save or create files for quick code execution.

- **Disadvantages:**

- Not ideal for running large programs.
- Requires manually saving work if you want to retain results.
- Code can't easily be reused across different projects.

Practical Tools for Running Python Software

JupyterLab Notebooks

A web-based interactive development environment for writing and executing Python code.

- **Step 1: If necessary, install JupyterLab with pip.**

```
pip install jupyterlab
```

- **Step 2: Launch JupyterLab.**

Open the command line and run the following command to start JupyterLab:

```
jupyter-lab
```

- JupyterLab will launch in the present working directory of your terminal.
- **Step 3: Create a New Notebook.**

Once JupyterLab is open in your browser, click on the "Python 3" option under the "Notebook" section to create a new Python notebook.

Practical Tools for Running Python Software

JupyterLab Notebooks

- **Step 4: Run Code in Cells.**

In a Jupyter notebook, code is written in cells. Type the following in the first cell:

```
print("Hello, World!")
```

- To execute the code, press `Shift + Enter` or click the "Run" button.

- **Step 5: Use Markdown for Documentation.**

You can also create markdown cells for documentation. Switch a cell to markdown mode by selecting ***Markdown*** from the dropdown and typing text for documentation.

Practical Tools for Running Python Software

JupyterLab Notebooks

- **Advantages:**

- Interactive environment perfect for data science, research, and documentation.
- Supports code, markdown, and rich media in a single interface.
- Excellent for visualization and step-by-step execution.

- **Disadvantages:**

- More resource-intensive compared to command-line execution.
- Not ideal for large, standalone programs.
- Can be cumbersome for version control and collaboration.

Practical Tools for Running Python Software

Spyder IDE

An integrated development environment (IDE) specifically tailored for scientific computing with Python.

- **Step 1: Install Spyder (if not already installed).**

You can install Spyder using pip:

```
pip install spyder
```

- **Step 2: Launch Spyder.**

Open the command line and run the following command to start Spyder:

```
spyder
```

Practical Tools for Running Python Software

Spyder IDE

An integrated development environment (IDE) specifically tailored for scientific computing with Python.

- **Step 3: Create or Open a Python Script.**

Once Spyder is open, you can create a new Python script by going to **File > New File** or open an existing one using **File > Open**. Type the following in the script editor:

```
print("Hello, World!")
```

- **Step 4: Execute the Python Script.**

To run the script, press **F5** or click the "Run" button in the toolbar. The output will appear in the "Console" panel at the bottom.

- **Step 5: Explore Spyder's Features.**

Spyder offers many useful features, such as variable explorer (view and edit variables), integrated help, and an interactive IPython console.

Practical Tools for Running Python Software

Spyder IDE

- **Advantages:**

- **Integrated Development Environment:** Includes an interactive Python console, variable explorer, and script editor in a single interface.
- **Built-in Visualization Tools:** Direct integration with popular libraries like Matplotlib, allowing real-time plotting and visualization.
- **Interactive Debugging:** Provides robust debugging tools, including breakpoints and stepping through code to help with troubleshooting.

- **Disadvantages:**

- **Resource-Intensive:** Requires more memory and CPU compared to simpler text editors or the command line.
- **Not Ideal for Large Projects:** While great for small to medium scripts, Spyder might not be the best choice for managing large-scale software projects.
- **Less Customizable:** Compared to IDEs like VS Code or PyCharm, Spyder is less customizable.

Functions and Modules

Introduction to Functions in Python

What are Functions?

- **What are Functions?**

- A function is a block of organized, reusable code that performs a specific task.
- Functions allow you to encapsulate code logic, making it easier to call the same code multiple times without rewriting it.
- They are defined using the `def` keyword followed by the function name and parentheses `()`.

- **Example of a Simple Function:**

```
def greet(name):  
    print(f"Hello, {name}!")  
  
greet("Alice")
```

Introduction to Functions in Python

What are Functions?

- **Benefits of Using Functions:**

- **Code Reusability:**

- Functions allow you to write code once and reuse it wherever needed, avoiding repetition.

- **Modularity:**

- By breaking down complex problems into smaller functions, you make your code more organized and easier to understand.

- **Maintainability:**

- Functions allow you to easily update or debug parts of your code without affecting the rest of the program.

Introduction to Functions in Python

What are Functions?

- **Readability:**

- Well-named functions make the code more readable by summarizing what specific sections of the code do.

- **Avoiding Global Variables:**

- Functions help avoid the misuse of global variables by encapsulating variables locally within the function.

Introduction to Functions in Python

Functions with Return Values

- **What are Functions?**

- In addition to performing tasks, functions can return data back to the caller using the `return` statement.
- This allows functions to output a result or value that can be used elsewhere in your code.
- After a `return` statement, the function stops execution and passes the result back.

- **Example of a Function that Returns a Value:**

```
def add(a, b):  
    return a + b  
  
result = add(3, 4)  
print(result) # Output will be 7
```

Introduction to Functions in Python

Functions with `*args` in Python

- **What are `*args` in Functions?**

- `*args` allows you to pass a variable number of positional arguments to a function.
- The `*` operator collects all extra positional arguments passed to the function and bundles them into a tuple, making it easy to iterate over them or process multiple inputs.
- This makes functions more flexible, allowing them to accept varying numbers of arguments without needing to be rewritten.

```
def add_all(*args):  
    return sum(args)  
  
result = add_all(1, 2, 3, 4)  
print(result) # Output will be 10  
  
result = add_all(1, 2)  
print(result) # Output will be 3
```

Introduction to Functions in Python

Functions with `**kwargs` in Python

- **What are `**kwargs` in Functions?**
 - `**kwargs` allows you to pass a variable number of keyword arguments to a function.
 - The `**` operator collects these arguments into a dictionary.
 - Functions then accept a varying number of keyword arguments without needing predefined parameters.

```
def print_info(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")
```

```
print_info(name="Alice", age=30)
```

```
# Output:
```

```
# name: Alice
```

```
# age: 30
```

Documenting Your Functions

Documenting Your Functions

Numpydoc

- **What is Numpydoc?**

- Numpydoc is a docstring format used for documenting Python code, particularly in scientific computing and data science libraries.
- It's widely used in projects like NumPy, SciPy, and other scientific Python packages.

- **Structure of a Numpydoc Docstring:**

- Numpydoc follows a specific structure with defined sections to make documentation clear and consistent.
- Common sections include:
 - **Parameters:** Describes the function's inputs.
 - **Returns:** Explains the output of the function.
 - **Raises:** Lists exceptions that the function can raise.

Documenting Your Functions

Numpydoc

■ Example of a Numpydoc Docstring:

```
def add_numbers(a, b):  
    """ Add two numbers.  
  
    Parameters  
    -----  
    a : int  
        The first number.  
    b : int  
        The second number.  
  
    Returns  
    -----  
    int  
        The sum of the two numbers.  
  
    Examples  
    -----  
>>> add_numbers(3, 4)  
7
```

Documenting Your Functions

Numpydoc

- **Why Use Numpydoc?**
 - Provides structured, readable documentation that is easy to follow.
 - Helps automate documentation generation for large projects.
 - Widely recognized in the scientific Python community, promoting consistency.

Documenting Your Functions

Inline Type Hinting

- **What is Inline Type Hinting?**

- Inline type hinting allows you to specify the expected types of function arguments and return values directly in the function signature.
- Introduced in Python 3.5, type hints improve code readability and help developers understand what types are expected.
- Type hinting does not enforce type checking but provides a form of documentation for your code.

- **Example of a Function with Type Hinting:**

```
def add_numbers(a: int, b: int) -> int:  
    return a + b
```

Documenting Your Functions

Inline Type Hinting

- **Benefits of Using Inline Type Hinting:**

- **Improved Readability:**

- Developers can quickly understand what types are expected for arguments and return values.

- **Better Tooling:**

- IDEs and code editors can offer better auto-completion, static analysis, and error detection based on type hints.

- **Facilitates Collaboration:**

- Team members working on the same codebase can understand function signatures at a glance, improving collaboration and reducing bugs.

Modules

Modules

What are Modules?

- **Definition:**

- A module is a Python file containing Python code (functions, variables, classes) that can be reused in other programs.

- **Purpose:**

- Modules allow you to break your code into smaller, reusable, and manageable parts, promoting modular programming.

Modules help organize code and allow for code reuse across multiple programs.

Modules

Built-In Modules

- **The `import` Statement:**

```
import math
print(math.sqrt(16)) # Outputs: 4.0
```

- **Selective Imports (`from module import`):**

```
from math import sqrt
print(sqrt(16)) # Outputs: 4.0
```

- **Aliases (`import module as`):**

```
import numpy as np
print(np.array([1, 2, 3]))
```

Python allows flexible importing: full modules, specific parts, or using aliases for convenience.

Modules

Built-In Modules

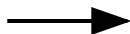
- Python comes with many useful built-in modules. Some examples include:
 - **math:** For mathematical operations.
 - **os:** For interacting with the operating system.
 - **random:** For generating random numbers.
- Example:

```
import random
print(random.randint(1, 10)) # Outputs a random number between 1 and 10
```

Modules

Built-In Modules

- Built-in modules provide essential functionality without requiring external installation.
- A list of built-in modules available to you can be found at <https://docs.python.org/3/py-modindex.html>



Python Module Index

[_](#) [a](#) [b](#) [c](#) [d](#) [e](#) [f](#) [g](#) [h](#) [i](#) [j](#) [k](#) [l](#) [m](#) [n](#) [o](#) [p](#) [q](#) [r](#) [s](#) [t](#) [u](#) [v](#) [w](#) [x](#) [z](#)

`__future__`

Future statement definitions

`__main__`

The environment where top-level code is run. Covers command-line interactive behavior, and ``__name__ == '__main__'``.

`_thread`

Low-level threading API.

`_tkinter`

A binary module that contains the low-level interface to Tcl/Tk.

a

`abc`

Abstract base classes according to :pep: `3119`.

`argparse`

Command-line option and argument parsing library.

`array`

Space efficient arrays of uniformly typed numeric values.

`ast`

Abstract Syntax Tree classes and manipulation.

`asyncio`

Asynchronous I/O.

`atexit`

Register and execute cleanup functions.

b

`base64`

RFC 4648: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85

`bdb`

Debugger framework.

`binascii`

Tools for converting between binary and various ASCII-encoded binary representations.

`bisect`

Array bisection algorithms for binary searching.

`builtins`

The module that provides the built-in namespace.

`bz2`

Interfaces for bzip2 compression and decompression.

Modules

Creating Custom Modules

- **Creating a Module:**

- You can create your own module by writing Python code in a `.py` file.

- **Example:**

- Create a file `my_module.py` :

```
def greet(name):  
    return f"Hello, {name}!"
```

- Use the module in another script:

```
import my_module  
print(my_module.greet("Alice")) # Outputs: Hello, Alice!
```

Custom modules allow you to structure your code and reuse it across different programs.

Modules

The Importance of `__init__.py`

- **What is `__init__.py` ?**
 - `__init__.py` is a special Python file used to mark a directory as a Python package.
 - Without this file, Python will not treat the directory as a package, and the modules inside the directory cannot be imported.
 - When a directory contains an `__init__.py` file, you can import modules from the directory like a package.
 - The file can be empty or contain initialization code for the package.

Modules

The Importance of `__init__.py`

- **Basic Example of Package Structure:**

```
my_package/  
  __init__.py  
  module1.py  
  module2.py
```

- You can import modules from the package:

```
from my_package import module1
```

`__init__.py` is by default empty, but can also be used to initialize package-level variables, import additional submodules, etc.

`__init__.py` is essential for structuring Python packages, enabling module imports, and controlling package initialization behavior.

Modules

Exploring Module Search Path (`sys.path`)

- **How Python Finds Modules:**

- Python searches for modules using the paths listed in `sys.path`.

- **Checking** `sys.path` :

```
import sys
print(sys.path)
```

- **Modifying** `sys.path` :

- You can add directories to `sys.path` to include custom modules from other locations:

```
sys.path.append('/path/to/custom/modules')
```

Python's module search path (`sys.path`) determines where Python looks for modules and can be modified to include custom directories.

Modules

Third-Party Modules and PyPI

- **Installing Modules via `pip` :**

- You can install external Python packages using `pip` :

```
pip install requests
```

- **Popular Third-Party Libraries:**

- `requests` : For making HTTP requests.
- `numpy` : For numerical computing.
- `pandas` : For data analysis.

- **PyPI (Python Package Index):**

- PyPI is the repository for sharing and downloading Python packages.

Third-party modules from PyPI extend Python's functionality, providing solutions for many domains.

Modules

Understanding `if __name__ == "__main__":`

- **What is `__name__` ?**

- In Python, `__name__` is a special built-in variable that represents the name of the current module.
- When a Python file is run directly, `__name__` is set to `"__main__"`. However, when a file is imported as a module, `__name__` is set to the module's name instead.

- **Why use `if __name__ == "__main__":` ?**

- This statement ensures that a block of code is only executed when the script is run directly, not when it's imported as a module in another script.
- It is commonly used to encapsulate the "entry point" of a Python script.

Modules

Understanding `if __name__ == "__main__"`

- **Example:**

```
# myscript.py
def main():
    print(5+5)

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

- **Behavior:**

- **When run directly:** The `main()` function will execute.
- **When imported by another file:** The `main()` function will not automatically execute.

It allows for reusable code by preventing specific parts of a script from running when the script is imported as a module elsewhere.

Modules

Benefits of Using Modules

- **Code Reusability:**

- Reuse code across multiple programs without rewriting it.

- **Organized Codebase:**

- Keep code clean and organized by dividing it into smaller modules.

- **Maintainability:**

- Easier to maintain and update your codebase with modular components.

Modules improve code organization, reusability, and maintainability, making it easier to manage larger projects.

Activity

Create and Use Your Own Python Module

Activity

Create and Use Your Own Python Module

Goals

- Create a Python module with two simple functions.
- Import the module into another script and use the functions.
- Properly numpypdoc and inline type hint your functions.

Activity

Create and Use Your Own Python Module

Steps:

1. Create a Python Module:

- In your working directory, create a new file called `my_module.py`.
- Add two simple functions to the file:

```
# my_module.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Activity

Create and Use Your Own Python Module

2. Write a Script to Use Your Module:

- In the same directory, create another Python file called `use_module.py`.
- Import your module and use its functions:

```
# use_module.py

import my_module

result1 = my_module.add(5, 3)
result2 = my_module.subtract(10, 4)

print(f"Addition: {result1}")      # Output: Addition: 8
print(f"Subtraction: {result2}")  # Output: Subtraction: 6
```

Activity

Create and Use Your Own Python Module

3. Run the Script:

- In the command line or terminal, run the `use_module.py` script:

```
python use_module.py
```

Dependency Management and Environment Requirements

Clean, Reproducible Code

Environment Management with Anaconda

Creating Reproducible Development Environments

- An environment is like a tissue culture flask.
- It guarantees that the software in one project doesn't interfere with another.
- This enables a stable and reproducible space for your code.

Environment A
Game Dev



Python 3.7
pygame 2.5.2

Environment B
Data Science



Python 3.7
pandas 2.2.0
numpy 1.26.4

Environment C
Web Dev



Python 3.6
django 2.2
beautifulsoup4 4.12.3

Environment Management with Anaconda

Why Does it Matter?

Version Inconsistencies.

- Python libraries and tools are constantly evolving.
- Different projects might require different versions of the same library, leading to conflicts and unexpected behavior.

Reproducibility.

- For scientific computing and data analysis tasks, it's crucial to reproduce results.
- This is challenging without a consistent environment, especially when sharing work with peers or publishing results.

Environment Management with Anaconda

Why Does it Matter?

Ease of Sharing.

- With a well-managed environment, developers can easily share their projects, ensuring that others can run their code without stumbling upon missing dependencies or version issues.

Isolation.

- Keeping project environments separate ensures that specific dependencies or version requirements of one project don't interfere with another, leading to cleaner and more stable software.

Environment Management with Anaconda

Miniconda vs Anaconda

Size and Content.

- Anaconda is a large distribution that comes pre-loaded with over 1500 packages tailored for scientific computing, data science, and machine learning.
- Miniconda, on the other hand, is a minimalistic distribution, containing only the package manager (conda) and a minimal set of dependencies.
- Due to its bundled packages, Anaconda requires more disk space upon installation compared to Miniconda.

Flexibility vs. Convenience.

- While Anaconda provides an out-of-the-box solution with a wide array of pre-installed packages, Miniconda offers flexibility by allowing users to install only the packages they need, helping to keep the environment lightweight.

Environment Management with Anaconda

What is a Package?

Software Collection.

- A package is a bundled collection of software tools, libraries, and dependencies that function together to achieve a specific task or set of tasks.

Version Management.

- Each package has specific versioning, allowing users to install, update, or rollback to particular versions as needed, ensuring compatibility and stability in projects.

Dependency Handling.

- When a package is installed in Anaconda, the system automatically manages and installs any required dependencies, ensuring seamless functionality and reducing manual setup efforts.

Environment Management with Anaconda

Creating a New Environment

Create a new environment.

```
conda create --name EnvironmentName
```

Create a new environment with specific Python version

```
conda create --name EnvironmentName python=3.10
```

Create a new environment from a YAML or text file.

```
conda create --name EnvironmentName file=package_contents.yml  
conda create --name EnvironmentName file=package_contents.txt
```

Environment Management with Anaconda

Creating New Environments from Text Files

Each package, and all of its dependencies, explicitly imported from a package manager (pip, conda, etc.)

Provides:

- Version Control (e.g., pyserial==3.5)
- Platform Control (e.g., sys_platform == "darwin")

```
[project]
name = "navigate-micro"
description = "Open source, smart, light-sheet microscopy control software."
authors = [{name = "The Dean Lab, UT Southwestern Medical Center"}]
readme = "README.md"
license = {file = "LICENSE.md"}
dynamic = ["version"]
classifiers = [
    "Intended Audience :: Developers",
    "Operating System :: POSIX :: Linux",
    "Operating System :: MacOS :: MacOS X",
    "Operating System :: Microsoft :: Windows",
    "Programming Language :: Python :: 3.9",
    "Programming Language :: Python :: Implementation :: CPython",
    "Programming Language :: Python :: Implementation :: PyPy",
]

requires-python = ">=3.9.7"
dependencies = [
    'matplotlib-inline==0.1.3',
    'PyYAML==6.0',
    'pyserial==3.5',
    'PIPython==2.6.0.1',
    'nidaqmx==0.5.7',
    'tiffiff==2021.11.2',
    'scipy==1.7.3',
    'pyusb==1.2.1',
    'pandas==1.3.5',
    'pandastable==0.12.2.post1',
    'opencv-python==4.5.5.62',
    'numpy==1.22.0; sys_platform != "darwin"',
    'numpy==1.21.6; sys_platform == "darwin"',
    'scikit-image==0.19.1',
    'zarr==2.14.2',
    'fsspec==2022.8.2; sys_platform != "darwin"',
    'fsspec==2022.5.0; sys_platform == "darwin"'
]
```

Environment Management with Anaconda

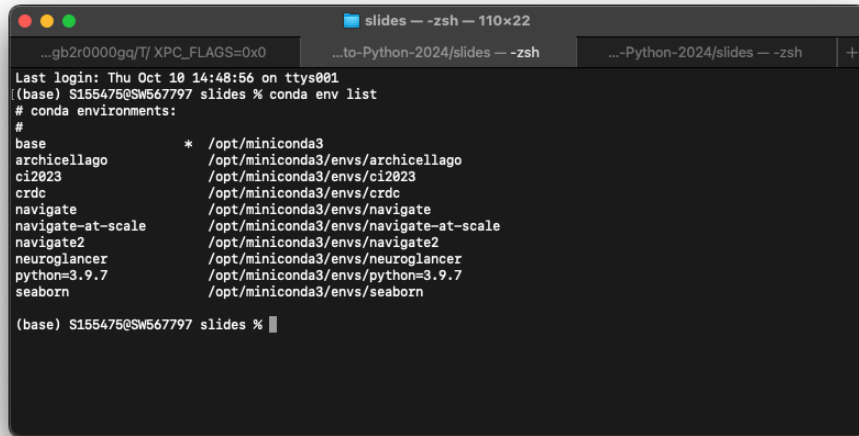
Working with Environments

List all environments.

```
conda env list
```

Activate an environment

```
conda activate EnvironmentName
```

A terminal window titled 'slides -- zsh -- 110x22' showing the output of the 'conda env list' command. The output lists several environments: 'base' at '/opt/miniconda3', 'archicellago' at '/opt/miniconda3/envs/archicellago', 'ci2023' at '/opt/miniconda3/envs/ci2023', 'crdc' at '/opt/miniconda3/envs/crdc', 'navigate' at '/opt/miniconda3/envs/navigate', 'navigate-at-scale' at '/opt/miniconda3/envs/navigate-at-scale', 'navigate2' at '/opt/miniconda3/envs/navigate2', 'neuroglancer' at '/opt/miniconda3/envs/neuroglancer', 'python=3.9.7' at '/opt/miniconda3/envs/python=3.9.7', and 'seaborn' at '/opt/miniconda3/envs/seaborn'. The prompt is '(base) S155475@SW567797 slides %'.

```
...gb2r0000gq/T/ XPC_FLAGS=0x0 ...to-Python-2024/slides -- -zsh ...-Python-2024/slides -- -zsh +
Last login: Thu Oct 10 14:48:56 on ttys001
(base) S155475@SW567797 slides % conda env list
# conda environments:
#
base                * /opt/miniconda3
archicellago         /opt/miniconda3/envs/archicellago
ci2023               /opt/miniconda3/envs/ci2023
crdc                 /opt/miniconda3/envs/crdc
navigate             /opt/miniconda3/envs/navigate
navigate-at-scale    /opt/miniconda3/envs/navigate-at-scale
navigate2            /opt/miniconda3/envs/navigate2
neuroglancer         /opt/miniconda3/envs/neuroglancer
python=3.9.7         /opt/miniconda3/envs/python=3.9.7
seaborn              /opt/miniconda3/envs/seaborn

(base) S155475@SW567797 slides %
```

Environment Management with Anaconda

Working with Environments

List all environments.

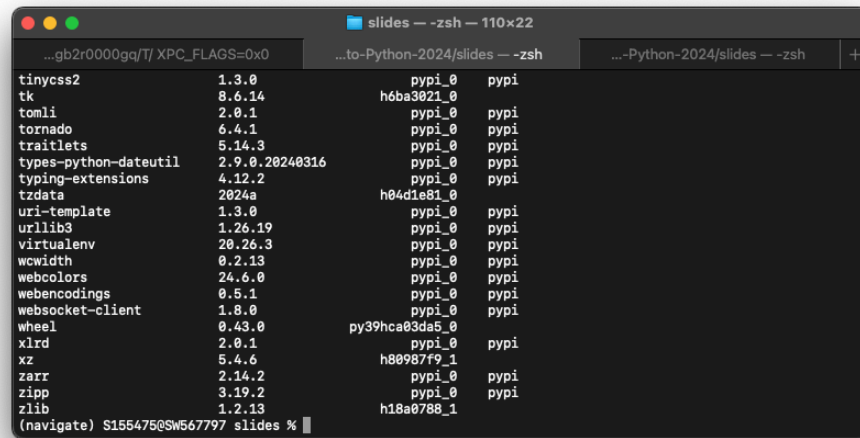
```
conda env list
```

Activate an environment

```
conda activate EnvironmentName
```

List Environment Packages.

```
conda list
```



A terminal window titled 'slides -- zsh -- 110x22' showing the output of the 'conda list' command. The output is a table of installed packages and their versions. The packages are listed in three columns, with the first column containing the package name, the second column containing the version number, and the third column containing the channel name. The packages are sorted by version number in descending order.

tinycss2	1.3.0	pypi_0
tk	8.6.14	h6ba3021_0
tomli	2.0.1	pypi_0
tornado	6.4.1	pypi_0
traitlets	5.14.3	pypi_0
types-python-dateutil	2.9.0.20240316	pypi_0
typing-extensions	4.12.2	pypi_0
tzdata	2024a	h04d1e81_0
uri-template	1.3.0	pypi_0
urllib3	1.26.19	pypi_0
virtualenv	20.26.3	pypi_0
wcwidth	0.2.13	pypi_0
webcolors	24.6.0	pypi_0
webencodings	0.5.1	pypi_0
websocket-client	1.8.0	pypi_0
wheel	0.43.0	py39hca03da5_0
xlrd	2.0.1	pypi_0
xz	5.4.6	h80987f9_1
zarr	2.14.2	pypi_0
zipp	3.19.2	pypi_0
zlib	1.2.13	h18a0788_1

(navigate) S155475@SW567797 slides %

Environment Management with Anaconda

Working with Environments

Important:

- Do not mix package managers (e.g., conda and pip).
- Be judicious and explicit with your dependencies.

Activity

Create a Python Environment and Install Packages

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

Objective:

- Create a new environment with a specific Python version.
- Activate the environment.
- Install dependencies from a text file.

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

Steps:

1. Create a New Environment with Python 3.9:

- Open your terminal or command prompt.
- Run the following command to create a new environment with Python 3.9:

```
conda create --name myenv python=3.9
```

2. Activate the New Environment:

- Once the environment is created, activate it using the following command:

```
conda activate myenv
```

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

3. Create a `requirements.txt` File:

- In your working directory, create a new text file called `requirements.txt` .
- Add a few packages to the file, for example:

```
numpy  
pandas  
matplotlib
```

4. Install Packages from the `requirements.txt` File:

- Use the following command to install the packages listed in the `requirements.txt` file:

```
pip install -r requirements.txt
```

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

5. Verify the Installation:

- After installation, verify that the packages were installed correctly by running:

```
conda list
```

6. Import Dependencies and Execute Logic:

- Create a Python script called `plot_image.py`.
- Import numpy and matplotlib.
- Create a random 2D matrix that is 512 x 512 pixels.
- Plot the original image.
- Manipulate the image (e.g., take the inverse).
- Plot the manipulated image.