Lecture 2:

- Python Basics

Homework:

- Quiz 1 on Tuesday (Lecture 1 concepts)
- Participation exercise #2 due Tuesday
The main features of Python:

- **dynamically typed**: variable types bound (and can change) at runtime
- **interpreted**: uses an underlying virtual machine implementation (CPython)
- **multi-paradigm**: supports OOP and functional constructs
- **unique syntax**: whitespace over punctuation (not a typical “C-like” language)
- **batteries included**: large number of libraries, etc.

In this class, we’ll use **miniconda** to help run and manage python:

- allows us to easily create python “**environments**”
- with specific python version and some built-in libraries
- comes with jupyter (for interactive notebooks)
- easy to install new libraries/modules we’ll use in class

To get started with **miniconda**:

1. After downloading and installing, first update:

    ```
    conda update -n base -c defaults conda
    ```

2. Create a conda environment: ... can have many environments

    ```
    conda create -y -n cpsc322 python=3.11
    ```

3. Activate the environment: ... depending on setup, each new shell

    ```
    conda activate cpsc322
    ```

4. If you want to deactivate the environment:

    ```
    conda deactivate
    ```
Some basic miniconda setup:

- To turn off default (base) activation on shell startup:
  ```
  conda config --set auto_activate_base false
  ```
- To turn off the current environment display at prompt:
  ```
  conda config --set changeps1 false
  ```
- To see a list of environments:
  ```
  conda env list
  ```
- To remove an environment (replace `<env>` with environment name):
  ```
  conda remove -n <env> --all
  ```

From within your activated miniconda environment

- Install dependencies (modules): ... can add at any time
  ```
  conda install -y jupyter pytest
  ```
- To run the Python REPL from the command line in environment:
  ```
  python
  ```
- To run jupyter notebook from command line in environment:
  ```
  jupyter notebook
  ```
- To run pytest from command line in environment:
  ```
  pytest
Programming in Python: Basics

Comments: denoted using #

```python
# this is a comment
# this is another comment
```

- comments can go after statements on same line
- no direct support for multiline comments

Identifiers (names): letter or underscore followed by letters, digits, underscores

- variety of naming conventions used (see: peps.python.org/pep-0008/#naming-conventions)
- generally lower_case_with_underscores for vars and functions (vs CamelCase)

Variable Declarations: take the form `var = expr`

Simple examples:

```python
x = 42      # x assigned literal value 42
y = 3.14    # y assigned different literal value
x = True    # note same x
z = y       # z holds copy of y
```

Notes:

- no explicit types
- vars can hold values of different types as program runs
**Data types:** literal values

- **Integers:** typical values, no fixed size (like in C++, Java, etc)
- **Floats:** double precision (64 bit) decimal values
- **Bools:** either True or False
- **NoneType:** the type of a None value
- **Strings:** single line, '...' or "...", or multiline '''...''' or """..."""

*Note:* unassigned multiline strings often used as multiline comments

```python
"""This is a
 multiline string being used
 as a comment"""
```

Can also create multiline strings this way:

```python
s = ('This is also a 
   'mulitline string but being assigned 
   'to a variable s')
```
Expressions: all the normal things you would expect

\[
\begin{align*}
\pi &= 3.14159 \\
\text{radius} &= 5.0 \\
\text{height} &= 3.4 \\
\text{volume} &= \frac{1}{3} \times \pi \times \text{radius}^2 \times \text{height} \\
\text{float_val} &= 3/4 \quad \# \text{ float division} \\
\text{int_val} &= 3 // 4 \quad \# \text{ int division (floor function)} \\
\text{float_val} &= 3 // 4.0 \quad \# \text{ floor of float coercion} \\
x &= \text{round}(\pi) \quad \# \text{ calls round function} \\
x &=+1 \quad \# \text{ shortcut for } x = x + 1
\end{align*}
\]

we will see more examples as we go ...

Basic Output: the built-in print() function

- `print()` \# prints a blank line (newline)
- `print(42)` \# output a single value on a line
- `print('a', 'b', 'c')` \# outputs first arg, space, second arg, etc

Using “format” strings \... see docs.python.org/3/tutorial/inputoutput.html

\[
\begin{align*}
x &= 42 \\
\text{print('the value is', } x) \\
\text{print('the value is {}\'.format(x))} \\
\text{print('the values are', } x, \ 'and', \ x + 1) \\
\text{print('the values are {} and {}\'.format(x, x + 1))} \\
\text{print(f'the value is {x}')}
\end{align*}
\]

Basic Input: the built-in input() function

- `answer = input()` \# prompts for input, returns string
- `answer = input('Enter something: ')` \# displays string as prompt

Examples:

>>> answer = input()
hello
>>> print(answer)
hello
>>> answer = input('Enter your name: ')
Enter your name: Alice
>>> print(answer)
Alice
Getting Help

from within python (REPL), can use the `help()` function

```python
>>> help(print)
Help on built-in function print in module builtins:

print(*args, sep=' ', end='
', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.
```

...
Type conversion

Built-in functions for casting (coerce values to types)

- \texttt{str(x)} converts \( x \) to a string (also \texttt{repr(x)})
- \texttt{int(x)} converts \( x \) to an integer
- \texttt{float(x)} converts \( x \) to a float
- \texttt{bool(x)} converts \( x \) to a bool (truthy/falsy values)
- ... plus others

\begin{verbatim}
>>> str(3.14)
'3.14'

>>> int('42')
42

>>> int(3.14)
3

>>> bool(42)
True

>>> bool(0)
False

>>> 42 and True
True

>>> float(42)
42.0

>>> float('42')
42.0

>>> int('abc')
...
ValueError: invalid literal for int() with base 10: 'abc'
\end{verbatim}
Can use the `type(x)` function to get the type of x

```python
>>> type(42)
<class 'int'>

>>> type('abc')
<class 'str'>

>>> type(3.14)
<class 'float'>

>>> type(False)
<class 'bool'>

>>> type(42) == int
True

>>> type(int)
<class 'type'>
```

Can use the `isinstance(x, t)` function to check type of x

```python
>>> isinstance(42, int)
True

>>> isinstance(42, float)
False

>>> isinstance('abc', str)
True

>>> isinstance(3.14, float)
True

>>> isinstance(3.14, int)
False

>>> isinstance(False, bool)
False

>>> isinstance(False, bool)
True
```
Python Programs and Modules

In this class, we’ll “run” our python programs in a couple of ways:

• As stand-alone program files from command line: `python prog.py`
• From jupyter notebooks (more later)

In python, every file can be used as a “module” (library)

• Often our programs will be multi-file ... with one main program file

Each program file consists of python code

• For us usually various functions and classes

A main program file will typically end with:

```python
if __name__ == '__main__':
    ... do something useful, like call a main() function ...
```
Participation Exercise #2:

1. Create a `cpsc322` conda environment
2. Make sure you can run python in the environment
3. Write a simple python program in a file `p1.py` that takes user input, contains at least one loop, has multiple if statements, and has print statements (so the program prints something based on the user input).
4. Run your program from the command line in your conda environment

For Tuesday, hand in a hardcopy with:

1. Your name
2. A copy of your program
3. Whether you had any issues setting up the environment, etc.