

■ About this class

- We meet on Tuesday and Thursday at 5:40pm in 6-125
- Class will be fast-paced
- Mixed focus on implementing math-related problems and practical tools

Syllabus

Course grading

- Problem sets: 40%
- Final project: 10%
- Midterm: 25%
- Final Exam: 25%

If your final exam grade is higher than your midterm exam grade, then I'll substitute your final exam grade for your midterm grade.

Exams will generally be cumulative in nature because we're going to continue building your base knowledge throughout the semester.

Grade breakdown

We'll follow the Baruch scale for grades so there shouldn't be any surprises.

- A: 93-100
- A-: 90-92.9
- B+: 87.1-89.9
- B: 83.0-87.0
- B-: 80.0-82.9
- C+: 77.1-79.9
- C: 73.0-77
- C-: 70.0-72.9
- D+: 67.1-69.9
- D: 60.0-67.0
- F: below 60.0


■ Syllabus

■ Problem sets

I just want to state some policies around our problem sets.

- Late submissions are penalized 5 points every 3 hours past the deadline.
 - Ex. If the deadline is 12am, but you submit at 6am, your max possible score is a 90 for the homework.
- All non-programming questions in problem sets should be typed up and submitted as a PDF.
- Feel free to collaborate on homework with your classmates, but you must note who in your homework submission.
 - We'll limit this to groups of 2!


 Syllabus

 Textbooks

There is no official textbook.

If you need a reference guide, then you should look into Python Crash Course by Eric Matthes.

 Syllabus

 Contact

If you need to contact me about any questions or concerns, you can reach me at jaimeabbariao@gmail.com

Just make sure that you prepend your email subject with **MTH 3300**.

■ Tips for doing well in this class

- Practice everyday
 - You will not do well on the exams if you do not practice coding
- Ask questions early and often
- Collaborate with your classmates

■ What is programming?

Programming is the process of creating instructions that a computer can follow to perform specific tasks.

A computer program follows an algorithm which is a set of instructions intended to solve some problem.

■ Pseudocode

We can express algorithms in pseudocode which is notation that resembles a programming language.

For example, let's say that we want to write a program that prints "Fizz" if a number is divisible by 3, "Buzz" if a number is divisible by 5, and "FizzBuzz" if the number is divisible by both 3 and 5.

How can we solve this with pseudocode?

```
if number is divisible by 3 and number is divisible by 4 then
    print "FizzBuzz"
else if number is divisible by 3 then
    print "Fizz"
else if number is divisible by 5 then
    print "Buzz"
end
```

The important aspect about pseudocode is that you're expressing your thoughts in a manner that is structurally similar to how you're going to write code.

This is going to be an important aspect during exams! If you forget the syntax for certain things, but you can show that you know how to express your ideas with pseudocode, you will earn points.

■ What is programming?

■ Programming languages

Obviously, we cannot run computer programs with just pseudocode.

In order to create computer programs that can be transformed into machine language (binary), we need a human-readable language.

■ Introduction to Python

Python was created by Guido van Rossum and released to the public in 1991.

Python has positioned itself as a great introductory programming language for the following reasons:

- Ease of learning: Python has a simple and readable syntax which is designed to be easy to understand and write.
- Relevance: Python is widely used in the industry.
- Enforces good habits: Python encourages good programming practices like readability and simplicity

Here's a quick example of what Python code looks like:

```
# This code prints the famous "Hello, World!" message
print("Hello, World!")

# This code calculates the sum of two numbers
a = 5
b = 10
sum = a + b
print("The sum is:", sum)
```

■ Introduction to Python

■ Installing Python

This class requires that we have at least python@3.10

■ For MacOS users

If you don't already have Homebrew, please following the instructions on the site (<https://docs.brew.sh/Installation>) to install it.

In your terminal, run the following command:

```
brew update  
brew install pyenv
```

Then after installing **pyenv**, run the following commands:

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc  
echo '[[ -d $PYENV_ROOT/bin ]] && export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zshrc  
echo 'eval "$(pyenv init - zsh)"' >> ~/.zshrc
```

Restart your terminal after this is done.

■ Introduction to Python

■ Installing Python

■ For MacOS (continued)

Once this is done, we need to make sure that the following build tools have been installed:

```
xcode-select --install  
brew install openssl readline sqlite3 xz zlib tcl-tk@8
```

Run the following command to make sure that you've successfully installed the correct version

```
pyenv install 3.12  
pyenv global 3.12
```

Now check if you have the proper installation by running: `python -v`

■ Introduction to Python

■ Installing Python

■ For Windows users

Visit the official download page (<https://www.python.org/downloads/>) and download Python 3.12

Run the installer and make sure to add Python to your path.

Run the following command to make sure that you've successfully installed the correct version

```
python --version
```

■ Introduction to Python

■ Installing Python

■ For Linux users

If you don't have at least Python 3.10 installed on your machine, you can do one of two things:

- Use your distribution's package manager to download the appropriate version
- You can download the source code from the Python site and build it. Handy link here if you need it (<https://docs.python.org/3/using/unix.html>)

■ Introduction to Python

■ Using the Python interpreter

Before we create our first files, the easiest way to run Python is by running the following command in your terminal.

```
python
```

Calling `python` opens up the interpreter which allows us to have an immediate feedback loop.

To exit the interpreter, you can type in `exit()` or use `ctrl + d`

Exercise: Use the interpreter to print out your name

```
print("Jaime")
```

■ Code editors

Here are a list of code editors that I would recommend for beginners:

- Visual Studio Code (<https://code.visualstudio.com/>) (recommended)
- PyCharm (<https://www.jetbrains.com/pycharm/>)
- Zed (<https://zed.dev/>)
- neovim (<https://neovim.io/>)

For those that want a simple setup, VSCode will be enough.

NOTE: I do prefer students to use code editors over Jupyter notebooks because you will need to upload actual Python files for your assignments

■ Primitive data types

We'll go over some fundamental data types that exist in Python.

There are going to be more later on, but for now, we'll go over the following:

- `int`: represents integers like 0, 1, 5, etc.
- `float`: represents real numbers like 3.14, 1.0, etc
- `bool`: represent Boolean values True and False
- `str`: represents a sequence of characters such as "your name"

There is also something we call the `NoneType` in Python which isn't a primitive data type but is special in that it denotes just one value: `None`

■ Variables

Variables exist as a mechanism to store information that we can potentially read or mutate within a computer program

Here are some examples of doing so:

```
name = "jaime"
ten = 10
not_true = False

sum_of_two_numbers = 1 + 1

name = "john" # we can rename variables as well
```

Exercise: Create the following variables and assign any value you want to it

- first_name
- last_name
- email

■ Variables

■ Best practices for naming variables

- Make sure your variable names denote the variable's intended purpose
- Variable names should be in snake case, i.e. `name_like_this`

■ Variables

■ Multiple inline assignments

You can assign values to different variables on the same line.

```
a_number, another_number = 1, 2
```

■ Swapping values between two variables

Suppose that you want to swap the values of two variables.

If you reason it out, you can do the following:

```
a, b = 10, 20  
  
temp = a  
a = b  
b = temp
```

But in Python, we can take advantage of the inline assignment of multiple variables!

```
a, b = 10, 20  
  
a, b = b, a
```

■ Variables

■ type()

We can check the type of a variable with the following function: `type()`

Run the following commands in your console:

```
type("name")  
type(12)  
type(3.15)  
type(True)  
type(False)  
type(None)
```

■ Type conversion

In Python, there are ways to coerce types to go from one to another. There are going to be times when you need a string to be an integer or an integer to be a string.

We'll explore this with the primitive types we know.

■ Type conversion

■ int -> str

If we want to convert an integer to a string, one way we can do so is by using the `str` function.

```
a_number = 42  
a_number_as_str = str(a_number) # Output: "42"
```

■ Type conversion

■ float -> str

If we want to convert a float to a string, one way we can do so is by using the `str` function.

```
a_number = 42.0  
  
a_number_as_str = str(a_number) # Output: "42.0"
```


■ Type conversion

■ bool -> str

If we want to convert a boolean to a string, one way we can do so is by using the `str` function.

```
a_bool = False  
a_bool_as_str = str(a_bool) # Output: "False"
```

■ Type conversion

■ int -> float

If we want to convert an integer to a float, one way we can do so is by using the `float` function.

```
an_int = 12  
  
an_int_as_float = float(an_int) # Output: 12.0
```

■ Type conversion

■ float -> int

If we want to convert an float to an integer, one way we can do so is by using the `int` function.

```
a_float = 12.2  
a_float_as_int = int(a_float) # Output: 12
```

Notice here that when we cast a float to an int, we lose the precision!

■ Type conversion

■ `str | int | float -> bool`

If we try to convert any of the primitive types to a boolean, we notice a funny pattern.

```
# what is the output the following?
```

```
bool("")  
bool("some_string")
```

```
bool(0.0)  
bool(12.0)  
bool(-0.2)
```

```
bool(1)  
bool(0)  
bool(-1)  
bool(100)
```

We say that some values are `truthy` or `falsy` based on their values.

In the above examples, we categorize an empty string `""`, `0`, and `0.0` as `falsy`.

We can then see that a string with more than character, non-zero integer and floating point values are considered `truthy`.

Basic input and output operations

print()

We can use the `print()` function to log data to the console.

```
a = 10
b = "name"
c = False

print(a)
print(b)
print(c)
```

f-strings

In Python, we use f-strings to interpolate variables into regular string expressions.

Take the following state for example:

```
name = "Jaime"
print(f"Hello, {name}")
```

When we run this on the console, we find that this prints out `Hello, Jaime`.

Exercise: Print out the sum of two variables `a` and `b` as `a + b = <sum>`. In other words, if `a = 1` and `b = 10`, then I should see the output as `1 + 10 = 11`

Basic input and output operations

input()

Suppose now that we want to get user input. Python has an `input()` function that allows us to do this.

```
name = input("What is your name?")  
  
print(f"My name is {name}")
```

The `input()` function returns a value of the type `str`

This means that if we wanted instead to ask the user for a number, we'd have to cast the `str` into an `int` like we've seen before.

```
name = input("What is your name?")  
age = int(input("How old are you?"))  
  
print(f"{name} is {age} years old")
```

■ Running your first Python file

■ add.py

```
a = int(input("Enter one number: "))  
b = int(input("Enter another number: "))  
  
print(a + b)
```

If want to run this code in the terminal, run your file with the following command:

```
python <path-to-file>/add.py
```