STATS 507
Data Analysis in Python

Week7: Midterm Recap

(A collections of previous recaps…)

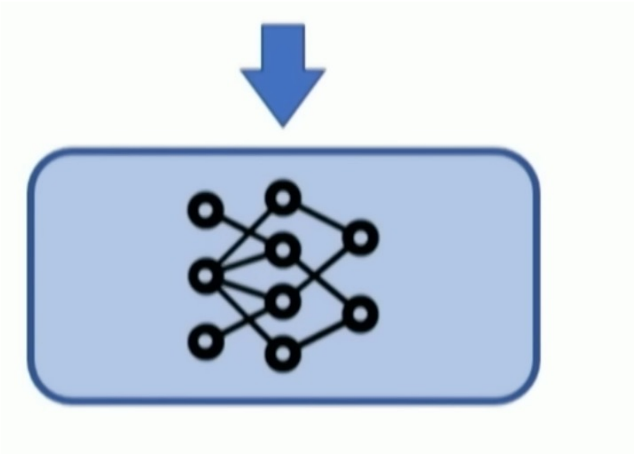
Recall our course goals

1. Establish a broad background in **Python** programming
2. Prepare you for the inevitable coding interview
3. Survey popular **tools** in academia/industry for data analysis
4. Learn how to read documentation and quickly get familiar with new tools.

The scope of midterm will be everything covered in lecture, in-class practice, HWs up to and including NumPy. The review session will just recap some of the concepts…

Generating Images from Natural Language

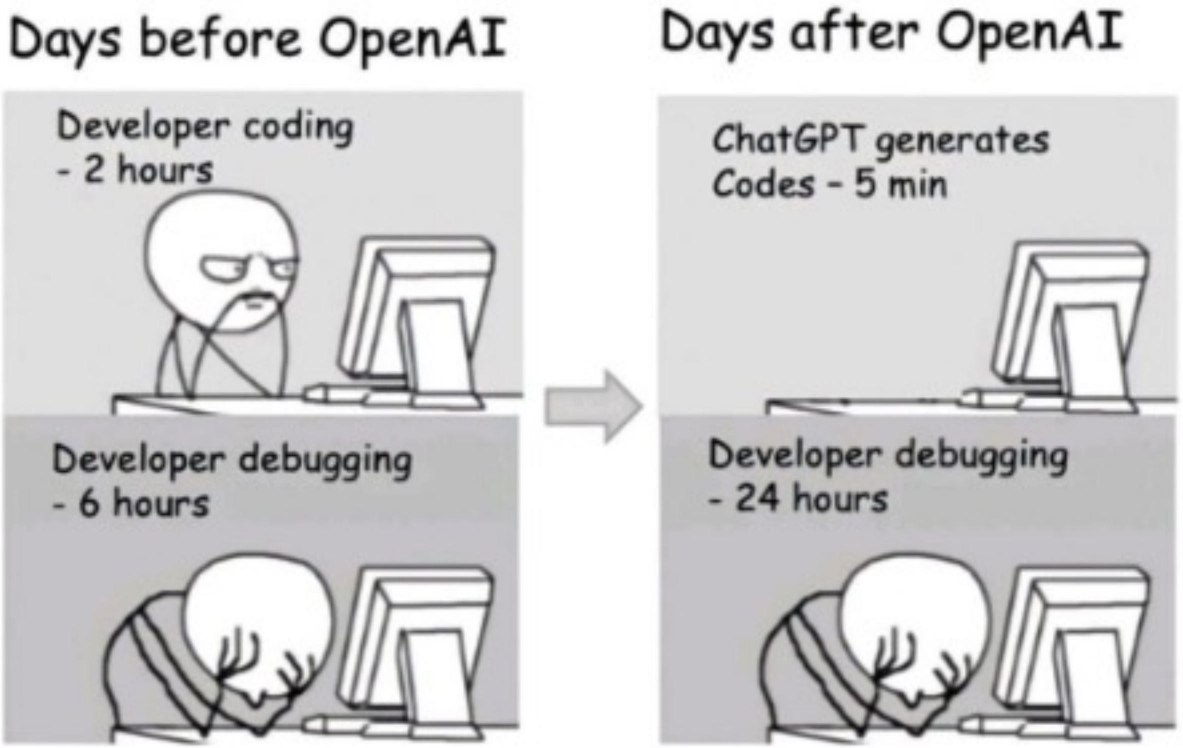
Prompt: “A cat walking a dog at University of Michigan central campus”



Generated by [DALL·E 3](): able to generate **new data**

Introduction

Big data/ deep learning is revolutionizing so many fields…



A great tool (used properly) for software engineer/data scientist/education
(The ethics guideline: https://genai.umich.edu/guidance/students)

Isaac Gym from Nvidia Isaac Sim

Robotics

Grading

Final grades will be based on

Assessment	Percentage
Homework (8)	40%
Midterm	30%
Final Project	30%

An **in-person** midterm is scheduled on :

Wednesday, 10/09/24 in AUD C AH

Consider it as a mini-interview for basic Python concepts…

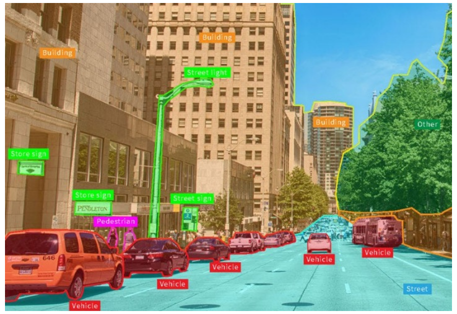
Final Projects (30%)

Assuming knowing Python and related tools, the final project for this course is to build ANY end-to-end project with [hugging face](#) where you can use any open-sourced dataset and model (pretrained or fine-tuned).

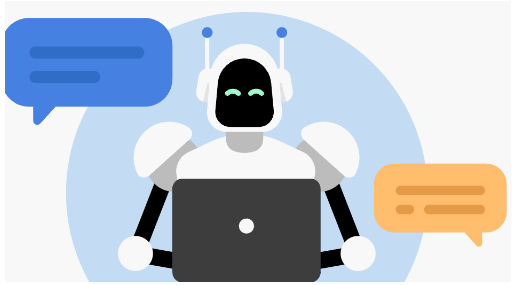
More details on submission guidelines and evaluation criteria will be provided with the project announcement.

Music generation

Computer Vision



Large Language Models



A general guideline

Time Limit: You have **80 mins** to complete this exam.

Total Points: This exam is worth **100** points total.

Question Format: The exam is **closed-book** and will consist of various question types, including multiple choice, short answers, and writing out the code outputs…

Clarity: Write legibly and explain your reasoning **clearly**. If we can't read it, we can't grade it.

Tips for preparation: ask what, why and how!

So far…

Write **correct** Python

- Use built-in objects
- Define our own class

Write **effective** Python (evaluate Python)

NumPy

What is Python?

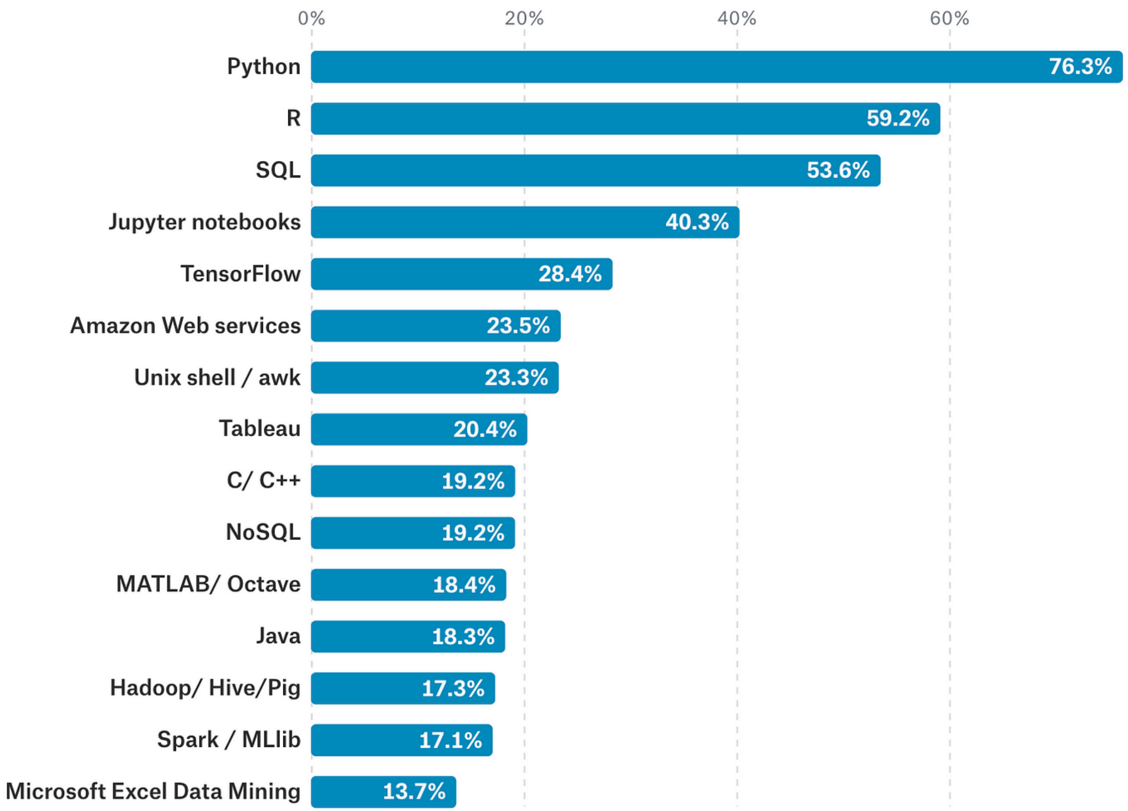
Python is a **dynamically typed**, **interpreted** programming language

- Created by Guido van Rossum and first released in 1991.
- Design philosophy **simple, readable**

Dynamically typed	Interpreted
In many languages, when you declare a variable, you must specify the variable's **type** (e.g., int, double, Boolean, string). Python does not require this, the type of a variable is defined at **runtime**. v.s. statically typed, flexible yet more error-prone	Some languages (e.g. C/C++ and Java) are compiled: we write code, from which we get a runnable grogram via **compilation**. In contrast, Python is **interpreted**: a program, called the **interpreter**, runs our code directly, line by line. v.s compiled: simple yet slower

Why Python?

Increasing, Python is the language of data science and general programming.



- Popular
- Easy to learn
- Open-sourced (no license needed)

Data science community Kaggle's annual survey "The State of Data Science& Machine Learning" asks the question "What tools are used at work?" (reference)

Object data types in Python

Different object can **represent** different concepts.

ANY object has a type that defines what kind of operations programs can do to them

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Ref:
https://www.w3schools.com/python/python_datatypes.asp

String and its operations

String is an **<u>immutable</u>** sequence of case sensitive characters.

- Letters, special characters, spaces, digits
- “me”, ‘States 507'
- Another built-in **date type** in Python

Create a string (single or double quote)

- str1 = “This is a string”
- str2 = ‘This is also be a string’

Manipulate a string

- Sequence indexing (how is it defined and done, potential error message…)
- Slicing (Syntax)
- Properties: immutability
- Other string methods and dot notation

Same for list, tuple and dictionaries

Lists are (mutable) sequences whose values can be of **any data type**

- We call those list entries the **elements** of the list

Mutable: We can **change** values of specific elements of a list.

Add new element, **delete** existing ones, **reorder(sort)** and many more…

Tuples are **immutable** sequences whose values can be of **any data type**

Dictionaries are …

Tips: think about what, how to create, operations, comparisons between different data types…

Recap: Class as programmer-defined types

Objects are instances of a class and are a data abstraction that captures:

- An **internal representation**
  - Through data attributes
- An **interface** for interacting with objects
  - Though methods (aka procedures/functions)
  - Defines behaviors but hides implementations

Creating the class (a parallel to function)

- Define the class **name**
- Define class data attributes
- Define procedural attributes

Using the class:

- Create new **instances** of the class
- Doing operations on the instances

Recap: Inheritance

Inheritance is perhaps the most useful feature of object-oriented programming

Inheritance allows us to <u>create new classes from old ones</u>

Parent class

(base class/superclass)

Child class

(derived class/ subclass)

- **Inherit** all data and behaviors of the parent class
- **Add** more **info(data)**
- **Add** more **behavior**
- **Override** behavior

Class definition Class name

Parent class

```
class Cat(Animal):
    def speak(self):
        print("meow")
```

```
my_cat = Cat(7)
my_cat.set_name("Fay")
print(my_cat.get_name())
print(my_cat.speak())
```

```
Fay
meow
None
```

Recap: Iterator as an object

An iterator is an object that represents a “data stream”

Supports method `__next__()`:

returns next element of the stream/sequence

raises `StopIteration` error when there are no more elements left

```
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
    def __iter__(self):
        return(self)
s = Squares()
for x in s:
    print(x)
```

Iterable means that an object has the `__iter__()` method, which returns an iterator. So `__iter__()` returns a new object that supports `__next__()`.

`__next__()` is the important point, here. It returns a value, the next square.

Now `Squares` supports `__iter__()` (it just returns itself!), so Python allows us to iterate over it.

Recap: Handling exceptions

Exception handler in Python

```
try:
    # do some potentially
    # problematic code
except:
    # do something to
    # handle the problem
```

```
if <all potentially problematic code succeeds>:
    # great, all that code
    # just ran fine!
else:
    # do something to
    # handle the problem
```

Besides `except` blocks

- `else`
  - Body will always be executed **when** `try` block competes with no exceptions
- `finally`
  - Useful for cleanup actions

```
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    else:
        return f"The result is: {result}"
    finally:
        print("Execution complete.")
```

```
Execution complete.
'The result is: 2.0'
```

Problems with timing and counting

Timing the exact running time of the program:

- Depends on **machine**
- Depends in **implementation**
- Small inputs don't show growth

Counting the exact number of steps:

- Gives a formula
- Do NOT depend on machine
- Depends on the implementation
- Also consider irrelevant operations for largest inputs
  - Initial assignment, addition

Goal:

- Evaluate algorithms (not implementation)
- Evaluate scalability just in terms of input size

Asymptotic growth: the order of growth

We can evaluate programs by

- Timer
- Count the operations
- Abstract notion of order of growth

Goal: Describe how run time grows as size of input grows

- Want to put a **bound** on growth
- Do **NOT** need to be precise: “order of ” not ”exact” growth
- Want to focus on terms that grows most rapidly
  - Ignore additive and multiplicative constants

This is called order of growth

- Use mathematical notions of “Big Oh(O)” and “Big Theta(Θ)”

Θ(x) Complexity Classes

- $\Theta(1)$: denotes constant running time
- $\Theta(\log n)$: denotes logarithmic running time
- $\Theta(n)$: denotes linear running time
- $\Theta(n \log n)$: denotes log-linear running time
- $\Theta(n^c)$: denotes polynomial running time
- $\Theta(c^n)$: denotes exponential running time
- $\Theta(n!)$: denotes factorial running time

What's NumPy

Open-sourced add-on modules for **numerical computing**

1) NumPy: **numerical** python, have multidimensional arrays
2) Optimized library for **matric and vector computation**
3) Makes uses of C/C++ subroutines and memory-efficient data structure
4) Building block for other packages: SciPy, Matplotlib, scikit-learn, scikit-image and provides fast numerical computations and high-level math functions

Ref: https://numpy.org/doc/stable/user/absolute_beginners.html

Why NumPy (v.s. built-in lists)

Python is very slow and NumPy are much more **efficient**

- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - NumPy takes ~0.03 seconds

Have more advanced mathematical functions, **convenient**

- Have mathematical operations applied directly to arrays
  - Linear algebra, statistical operations...

Broadcasting and vectorization saves time and amount of code

1. NumPy as numerical computing (Basics)

2. Array indexing

3. Vector and Matrix Operations

4. Broadcasting

Good luck!

Questions?