AI Planning: a Light Introduction to Python
(with pretty pictures!)
Tech 411.02 F25

Steve Wissow
UNH Computer Science
sjw@cs.unh.edu

What is Python?

- High-level, interpreted programming language
- Easy to learn, emphasizes readability
- Widely used in data science, web development, automation

Try Python Now in the ‘REPL’ (Read-Evaluate-Print-Loop)

In a new terminal window, at a shell prompt $, type `python`:

```
$ python
Python 3.11.14 <some other messages...>
>>>
```

Now you have a Python REPL prompt: >>>

Print Statements

Python uses `print()` to output text:

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

Variables

Variables store data:

```
x = 5
name = "Alice"
pi = 3.14159
```

No need to declare types explicitly.

String Interpolation

Use ‘f-strings’:

```
>>> print(f"Hello, my name is {name}, I have {x} computers but
    there is only one pi ({pi})")
Hello, my name is  Alice , I have 5 computers but there is only
     one pi (3.14159)
>>>
```

Basic Data Types

Python has several built-in types:

- Integers: 1, 42, -7
- Floats: 3.14, -0.001
- Strings: "hello", 'world'
- Boolean: True, False

Scripts!

Why:

- keep a record of what you did
- sharing code with others
- write code once, run many times on different data!
- easier to debug

How:

- exit REPL (keep terminal open):

```
>>> ^D
$
```

- open new text editor window, save as: `script.py`
- write some code; save
- run:

```
$ python script.py
```

If-Then-Else Control Flow

Conditional statements allow branching:

```
month = 11
if month >= 10:
    print("pumpkin spice season")
elif month == 1:
    print("hot chocolate season")
else:
    print("apple cider donut season")
```

Match-Case Control Flow

Another idiom for branching:

```
flavor = input()
match flavor:
    case "pumpkin spice":
        print("good")
    case "apple cider":
        print("better")
    case "chocolate":
        print("best")
    case _:
        print("no thank you")
```

...and `input()` to allow user of your script to supply (a small amount) of data

Loops

Loops let you repeat code:

```
for i in range(5):
    print(i)

x = 0
while x < 5:
    print(x)
    x += 1
```

Lists and Loops

Lists store multiple values:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Functions

Functions let you reuse code:

```
def greet(name):
    print("Hello", name)

greet("Alice")
greet("Bob")
```

Functions can accept arguments and return values.

Importing Modules

Use modules for extra functionality:

```
import math
print(math.sqrt(16))

import random
print(random.randint(1,10))
```

Python has many built-in and external modules.

Classes (Minimal Introduction)

Python supports basic object-oriented programming:

```
class Person:
    def __init__(self, name):
        self.name = name

    def introduce(self):
        print(f"Hi, my name is {self.name}").

p = Person("Alice")
print(p.name)
p.introduce()
```

Classes let you organize data and behavior.

Example Program

Combine functions, loops, and lists:

```
names = ["Alice", "Bob", "Charlie"]

def greet_all(names):
    for name in names:
        print("Hello", name)

greet_all(names)
```

Python Summary

- Python is easy to read and write
- Print statements and variables are building blocks
- Control flow: if-then-else, loops
- Functions and modules organize code
- Minimal classes support structured data

Practice writing small programs!
Then take a 5-min stretch break!

Variables and Data Structures for State Space Search

whiteboard graph example:

- initial state s_i: "(I'm at the) IOL."
- goal predicate $goal(s)$: "$s ==$ Campus Creamery?"
- successor function: $expand(s)$:

```
{
    hallway outside main door,
    E-Center work space,
    driveway outside window,
}
```

- OPEN list: ordered list of generated successors that have not yet been expanded
- CLOSED LIST: set of states we've already visited (to catch ourselves before going in circles)

Depth-First Search (DFS)

- implement OPEN as a Last-In-First-Out (LIFO) stack
- push, pop

Breadth-First Search (BFS)

whiteboard graph example:

- implement OPEN as a First-In-First-Out (FIFO) queue
- insert, extract

Best-First Search

whiteboard metric-space example (grid world):

- static evaluation function $h(s)$: the "value" of state s
- implement OPEN as a prority queue: *lowest-value element comes out first*
- insert, extract-min

Lab Time!

Finish implementing search code, and make some pretty pictures to visualize search behavior.

```
# clone lab repo
$ git clone http://github.com/sjwo/search-viz-lab-student

# create Python virtual environment
$ cd search-viz-lab-student
$ python -m venv .venv --prompt venv
$ source .venv/bin/activate

# new prompt!
(venv) $

# try to run
(venv) $ python main.py
```

Install Dependencies

stored in `pyproject.toml`
install with `pip` (downloads from pypi.org):

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

(still in `(venv)` from here on out)

Try to run again!

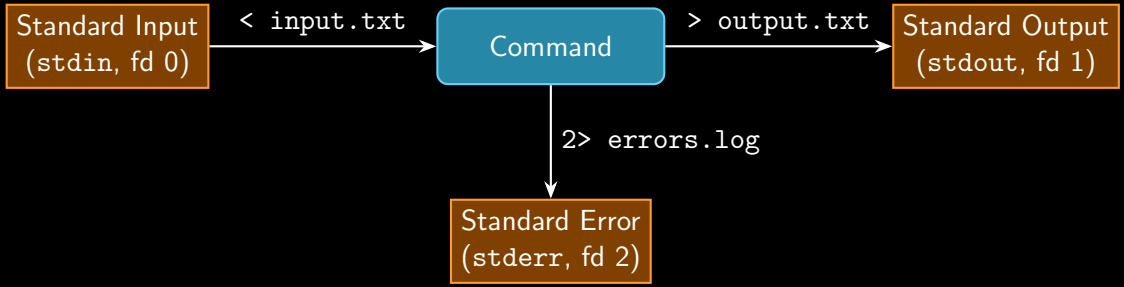
What's the problem now? How to fix it?

Usage

```
python main.py --help
```

Helpful information!

IO Redirection




```
command < in > out 2> err
```

Challenges:

- save problem instance to file
- run search on problem instance
- save search run expansions log to file
- visualize search run