0MR

Python for Robotics

Introduction

{“Jan Pohludka”: “216846@vutbr.cz”}

TODO

1. ECOSYSTEM (venv, pip, entry point, project structure)
2. DATA STRUCTURES (List, Numpy Array, Tuple, Set, Dict)
3. CLASSES (Inheritance, Abstraction, Polymorphism)
4. STATE SPACE SEARCH (DFS, BFS)

#1 ECOSYSTEM / venv

```
python3 -m venv .venv
```
use module -m venv with name .venv

```
.venv\Scripts\activate
```
```
(source .venv/bin/activate)
```
= run activation bash script

```
deactivate
```
= run deactivation bash script

#1 ECOSYSTEM / pip

ACTIVATE VENV FIRST!

`pip list`
= list all packages (global / current venv)

`pip install [name]`
= install package

`pip freeze > requirements.txt`
= save list of all packages to file

`pip install -r requirements.txt`
= installs all packages from the list

#1 ECOSYSTEM / entry point

```
if __name__ == “__main__”:
```
> runs only when called using **python** command

> program starts at the first line

#1 ECOSYSTEM / project structure

```
my_robot_project/
|── .venv/                   <-- Sandbox (We don't touch this)
|── src/                 <-- Source Code
    |── __init__.py      <-- Tells Python "This folder is a package"
    |── main.py          <-- The Entry Point (The "Run" button)
    └── sensor_lib.py        <-- A module
|── tests/                   <-- Unit tests (If you want to be fancy)
|── requirements.txt     <-- The Ingredient List
└── README.md                <-- Instructions
```

#2 DATA STRUCTURES / cheat sheet

Structure	**Syntax**	**Type**	**Mutable**	**Best for**
List	[1, “B”, 3]	heterogeneous	YES	stack, queue, ordered lists
Numpy	np.array([1])	homogeneous	YES	matrixes, vectors
Tuple	(1,2,”C”)	heterogeneous	NO	coordinates, return vals
Set	{1, 2}	unique objects	YES	is X in SET?
Dict	{“k”:”v”}	key: value	YES	JSON, config, parent map

#2 DATA STRUCTURES / hacks

```
for i, point in enumerate(waypoints):
```
- *enumerate* creates a counter

```
for dist, t in zip(sensors, times):
```
- parallel looping

```
x, y, _ = get_robot_state()
```
- unpacking returned tuple (and ignoring values)

```
readings_mm = [r * 1000 for r in readings_m]
```
- list comprehensions (one liners)

#3 CLASSES / basics

= blueprint for individual instances
- CLASS = CAD drawing, INSTANCE = 3D printed part

- self ~ *this* = keyword for specific instance of this class

```
class name:
    __init__(self):
        self.attribute = value
```

- **constructor** is called when creating instance (init)

#3 CLASSES / inheritance

```
class child(parent):
```

= superclass (parent) -> child classes
- can have more attributes or methods
- can override parent methods

```
def __init__(self, parent_attribues, new_attributes):
    super().__init__(parent_attributes)
    self.new_attribues = new_attributes
```

- constructor of superclass must be called!

#3 CLASSES / abstract classes and methods

```
from abc import ABC, abstract method
```

```
= a class we don't want to instantiate (too general)
- creating rules for child classes
    > what they have to implement
```

#4 STATE SPACE / data structures

LIST stack [] = “todo list”

SET visited {} = fast to check if x in visited

DICT parent_map {“k”:”k-1”} = breadcrumbs for backtracking

NUMPY ARRAY grid = big matrix, represents state space