Introduction To Python

Yet Another Programming Lecture?

Organizational Stuff

- 5 lectures (3 this semester, 2 next semester)
- 4 x 45 min + 15 minutes break per lecture
- Github Repository for assignments and slides
- “Interactive” lecture notes with code sandboxes: https://jspieler.netlify.app/teaching/introduction-to-python

Organizational Stuff (continued)

- If there are questions or something is not clear, ask at any time!
- Please bring a laptop or tablet with you
- You can help shaping this lecture

jspieler@outlook.de

General Recommendations

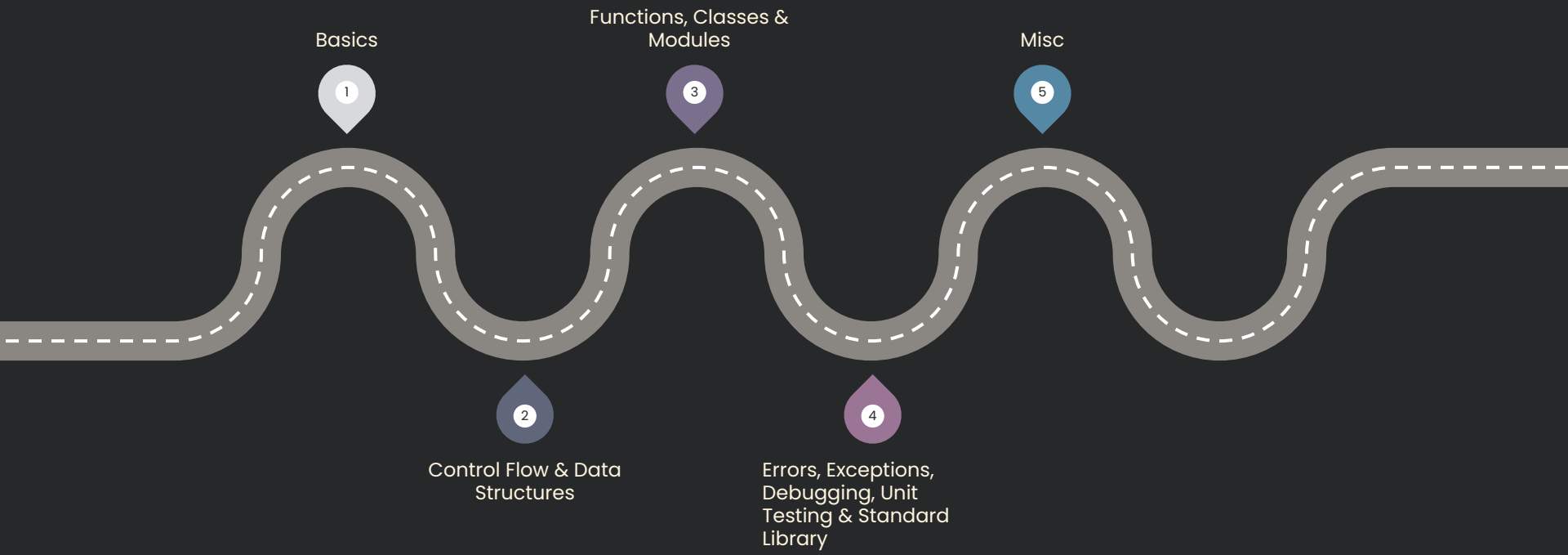
- Find a working mode that fits you best
- If you want to learn programming, practice it
- Get familiar with Git
- Start reading things

What This Lecture Is About

- Introduction to Python for beginners
- Requires basic programming knowledge
- Please tell me about your programming knowledge and expectations:

https://form.jotform.com/232300675128349

Lecture Overview



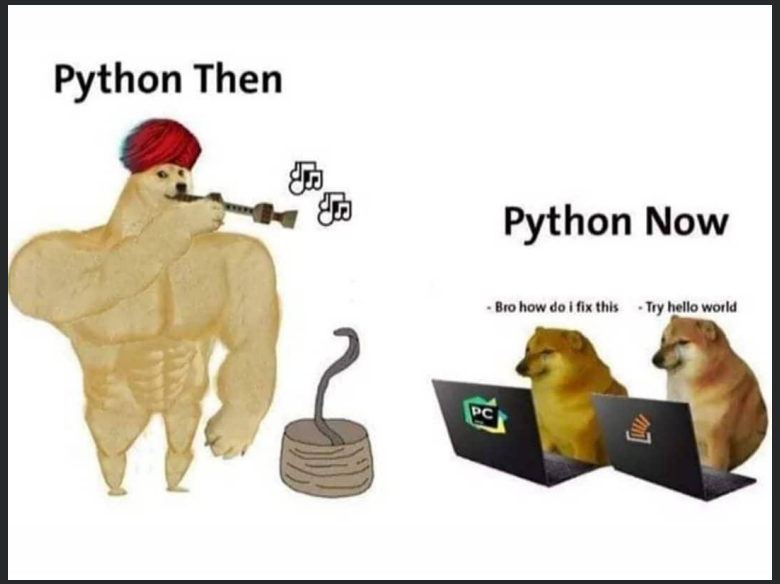
Introduction To Python

Let’s Dive Into The Basics

1 290 000 000

Whoa! That’s a huge number, but what is behind it?

Python?

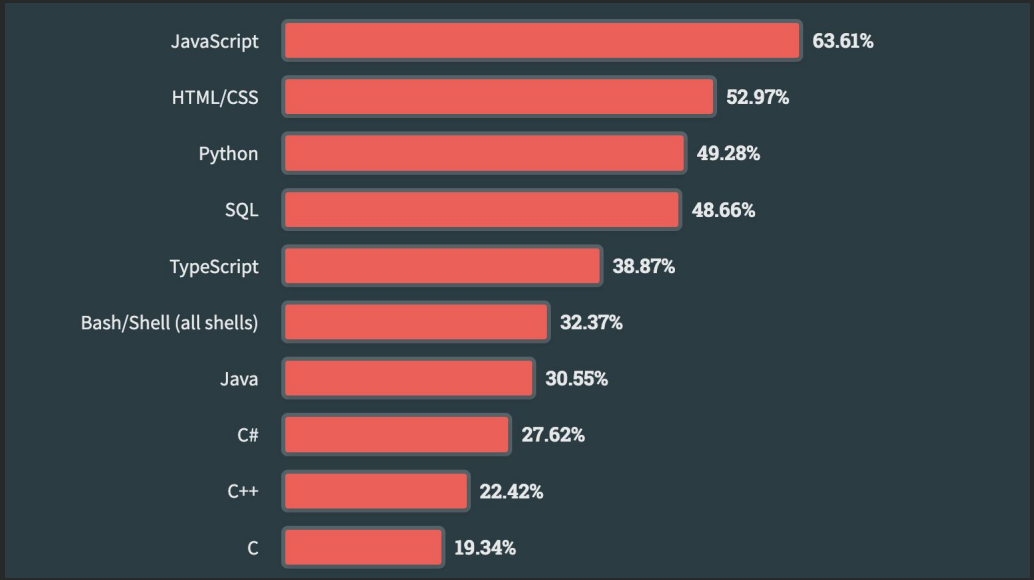


https://tinyurl.com/5fa28ru4

What Is Python?

- Python is a high-level programming language
- Developed by Guido van Rossum
- Released in 1991
- This lecture is about Python 3

Why Should I Learn Python?



https://tinyurl.com/3c2sjv4f

How To Run Python Code

- Interpreter needed
- Virtual environment recommended (we will talk about it in a second)

How To Invoke Python Code

- Shell/bash command line
- Create files using an editor or IDE (e.g. VS Code or PyCharm)

How We Will Code In This Lecture

- Please create a Github account if you don't have one yet
- Github repository for exercises:
  - Fork the repository
  - Create a new codespace
  - Use Git to commit your changes

Python Dependency Management

- Huge collection of libraries and packages
- Installation from Python Package Index via pip
- Relevant XKCD: https://xkcd.com/1987/
- Virtual environments are recommended to:
  - Resolve dependency issues
  - Create self-contained & reproducible projects
  - Avoid system pollution
  - Install packages without admin rights

Virtual Environments

- venv
- virtualenv
- Conda
- Docker

Basics

Let’s Start Programming

Variables

- Used for storing & managing data
- Can hold different data types
- Values can be assigned to variables using the assignment operator (=)
- Dynamic typing
- There are a few naming rules, recommended to adhere to conventions described in PEP 8

PEP?

- Stands for Python Enhancement Proposal
- Outline design decisions, standards and guidelines for Python
- Compliance with most PEPs is voluntary, but best practice
- Some of the fundamental PEPs are:
  - PEP 8 (Style Guide for Python Code)
  - PEP 257 (Docstring Conventions)

Data Types

- Numeric data types
- Strings
- Booleans
- None
- Data structures

Let’s fire up some Python console!

Numeric Data Types

- Integers
- Long integers
- Floating point numbers
- Complex numbers

String Operations

- Concatenation
- Formatting
- Repetition
- Indexing and Slicing
- Functions and methods

Booleans

```
loves_python = True
is_programmer = False

if loves_python:
    print("Python is great!")
```

None

- Special value used to define a null value or no value at all
- Not the same as 0, False or an empty string

Data Structures

- List
- Dictionary
- Tuple
- Set

Comments

- Document code & provide context
- Preceded by the # symbol
- Single-line and multi-line comments
- Different opinions on how and when to comment code among programmers

Operators

- Assignment Operators
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Identity & Membership Operators

One Important Thing About Assignments

- Python stores values in memory and associates them with their assigned variable names
- Python uses “pass by object reference” or “pass by assignment”:

```
a = [1, 2, 3]
b = a
a.append(4)
print(a)  # [1, 2, 3, 4]
print(b)  # what will this return?
```

Identity and Membership Operators

- Identity operators: `is` and `is not`
- Membership operators: `in` and `not in`
- Ternary operator:
  `[on_true] if [expression] else [on_false]`

Conditions & Loops

... What Else?

Conditions

```
temperature = 23

if temperature > 30:
    print("It's a hot day! Stay hydrated and wear sunscreen.")
elif 20 <= temperature <= 30:
    print("The weather is pleasant. Enjoy your day!")
elif 10 <= temperature < 20:
    print("It's a bit cool. Consider wearing a light jacket.")
else:
    print("It's cold outside. Bundle up and stay warm!")
```

For Loops

- Try to avoid loops like this:

```
numbers = [1, 2, 3, 4, 5]
for i in range(len(numbers)):
    print(numbers[i])
```

- Use `for in` and `enumerate` instead

While Loops

- Basic structure:

```
count = 1
while count <= 5:
    print(f"Count: {count}")
    count += 1
```

- “Infinite” loops

Exiting Loops

- Exit loop prematurely via `break` statement
- Skip current iteration and move to the next one via `continue` statement

Data Structures

How To Store And Manipulate Data

Lists

- Ordered collection, can contain different data types
- Created using square brackets:
  `numbers = [1, 2, 3, 4]`
- Elements can be access via square brackets and index
- Index starts at zero
- Mutable (can be changed after creation)

List Operators

- Indexing & Slicing
- Concatenation
- Repetition
- `in` Operator
- List comprehension
- Functions & Methods

Tuples

- Ordered collection, can store different data types
- Immutable (can’t be modified in place)
- Created by enclosing values within parentheses:

```
fruits = ("apple", "banana", "kiwi")
empty_tuple = ()  # or use tuple()
single_element = ("zero",)
```

Dictionaries

- Unordered collection, can store arbitrary objects
- Dictionaries are mutable
- Indexed by keys (any immutable object such as strings or tuples)
- Initialized using a comma-separated list of key/value pairs enclosed in curly braces

Sets

- Sets are unordered and its elements unique
- Elements can be objects of different types
- Sets can be modified, but elements must be immutable
- Created using curly braces or `set()`:

```
fruits = {"apple", "banana", "cherry", "apple"}
other_set = set("apple", "samsung", 123)
empty_set = set()
```

Functions & Modules

How To Organize Your Code

Functions

- Allow isolating sub-tasks
- Enable code reusability
- Python treats them as objects
- Defined using the `def` statement

```
def greet(name):
    """Greets a person."""
    print(f"Hello {name}!")
    return value (optional)
```

argument

docstring (optional)

function body

Variable Scopes

- A namespace is a mapping between object names and their memory addresses
- Local, enclosing and global variables
- Python follows LGB/LEGB rule

Named Arguments & Default Values

```
def info(name, age, job):
    print(f"Name: {name}, Age: {age}, Job: {job}")

info("John Doe", "Sales Manager", 44)
```

Variable Number of Arguments

- Variable length arguments
- Keyword-argument pairs

Functional Programming & Anonymous Functions

- `lambda` operator for anonymous functions
- `map()`
- `filter()`
- `reduce()`

Modules

- Collection of code grouping functions, classes & variables together according to functionality
- Allow for managing complexity and promote code reuse
- To use a module, we need to import it:

```
import my_module
from my_module import *
import my_module as mm
```

About if __name__ == “__main__”

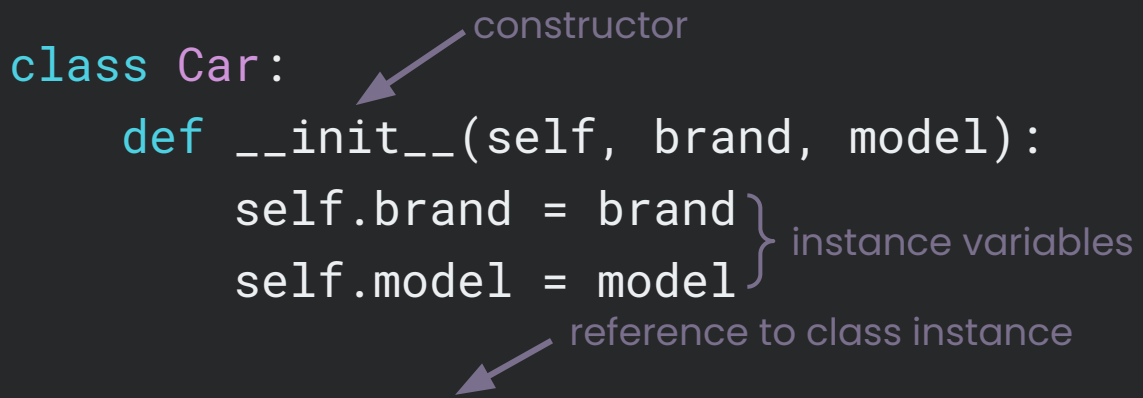
- One of the most viewed Stackoverflow questions
- Differentiate between code that is directly executed and code available for import from module

Object-Oriented Programming

What Is Object-Oriented Programming?

- Programming paradigm that uses objects and classes to structure code
- A class defines the attributes (data) and methods (functions)
- Enables reuse of code and managing of system complexity

Classes




```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def info(self):
        print(f"Type: {self.brand} {self.name}")

car = Car("Honda", "Accord")  # instantiate class
```

Protected & Private Attributes

- Not strictly enforced in Python, but best practice
- Indicates attribute should not directly be manipulated
- Helps improving code organization, maintainability & readability

More On Classes

- Class and instance variables
- Class methods and static methods

Fundamental OOP Concepts

- Inheritance: derived class inherits attributes and methods from parent class
- Encapsulation: provides interface for interaction with object while protecting integrity
- Polymorphism: provide single interface for entities of different types

OOP Example

encapsulation

inheritance

polymorphism

```python
class Shape:
    def __init__(self, color):
        self._color = color

    def area(self):
        pass

    def display(self):
        print(f"Shape color: {self._color}")
```

```python
class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self._radius = radius

    def area(self):
        return 3.14 * self._radius ** 2

    def display(self):
        super().display()
        print(
            f"Circle with radius {self._radius} "
            f"and area {self.area()}."
        )
```

Debugging, Error Handling & Unit Testing

Debugging

- Process of identifying & fixing bugs
- Essential skill
- Modern IDEs will support you

Errors & Exceptions

- Two types of errors in Python
  - Syntax errors
  - Exceptions
- Exception handling via `try` and `except`
- Avoid bare `except` clauses:

```
try:
    do_something()
except:
    print("Caught it!")
```

Use specific exceptions instead!

Common debugging Practices

1. Read error messages and tracebacks carefully
2. Isolate the problem
3. Reproduce the error
4. Track execution flow
5. Check data types
6. Check documentations

Fixing Bugs

- Understand the root cause
- Fix the actual bug
- Test your code
- Document changes

Unit Testing

- Smallest testable pieces of code (units) are tested for correctness
- Helps to isolate errors
- Two popular Python test frameworks:
  - unittest (standard library)
  - pytest

Unittest

- Test cases are created by subclassing `unittest.TestCase`
- Provides `TestLoader` and `TestSuite` to organize test suites
- Test fixtures & mocking

Standard Library

How To Use It

Python's Standard Library

- Built-in modules offering commonly used functionalities
- Distributed with Python installation
- Additional packages and frameworks can be installed for example from PyPi

Argparse

- Parse command-line arguments and options
- Simplifies taking inputs from command line
- Useful for scripts/program that require configuration or customization

Copy

- Create independent copy of objects
- Maintain integrity and prevent unintended side effects
- Shallow copies via `copy()` or deep copies via `deepcopy()`

File I/O

- Read and write files
- Recommended to use the `with` statement (context manager) to ensure files are properly closed after usage
- Exception handling is good practice

Logging

- Collect information, warnings and errors during execution of you program
- Logging can be configured:
  - Logging level
  - Formatting
  - Output directory
  - ...

Math

- Built-in module for mathematical operations
- Provides mathematical functions and constants

os

- Provides functions for interaction with operating system
- Allows you to perform various tasks related to
  - File and directory manipulation
  - Platform-independent file paths
  - Environment variables
  - Process management
  - …
- Exception handling may be needed

Pathlib

- Introduced in Python 3.4
- More intuitive and platform-independent way for working with file system paths and files
- Central concept is the `Path` object

Regex

- Regular expressions are powerful to search for and manipulate text data based on specific patterns
- Regular expressions consist of patterns that describe specific sequences of characters
- Regular expressions for common patterns can be found on the web
- There are also great (visual) regex testers like Debuggex

Sys

- Provides access to system-specific parameters and functions
- Often used to interact with Python runtime environment and system-related functionality
- Use `sys.exit()` to terminate scripts programmatically
- Standard input, standard output and standard error streams

Beyond The Basics

Equipping You for Your Further Python Journey

Type Annotations

- Introduced in Python 3.5
- Allow for specifying data types of variables, function parameters & return values
- Increase code readability and reduce type-related errors
- Not enforced at runtime by Python interpreter
- Use static type checkers like mypy
- Typing module documentation for more information

Decorators

- Modify or enhance function or method behavior without changing them
- Often used for logging, authentication, timing, etc.

```
def my_decorator(func):
    def wrapper():
        print("Inside decorator")
        func()
    return wrapper
```

```
@my_decorator
def say_hello():
    print("Hello!")
```

Dataclasses

- Introduced in Python 3.7
- Decorator-based
- Store and manage data
- Define classes with attributes, __init__, __repr__ and __eq__ automatically created
- Particularly useful for configurations or data objects

Enums

- Introduced in Python 3.4
- Define a set of named, constant values representing discrete choices or options
- Increase readability and maintainability
- Type-safe

How To Name Things In Code

- Avoid single letter variable names & abbreviations
- Don’t include types in variable names
- Include units in variable names
- Refactoring instead of “utils”
- “Consistency is key”: adhere to conventions and best practices

Python Best Practices

- Follow conventions to increase readability and maintainability
- Avoid magic numbers
- Comment and document effectively
- Use f-strings instead of string concatenation
- Avoid global variables

Best Practices (continued)

- Use context manager (`with`) instead of `try` and `finally`
- Avoid bare `except` clauses
- Use comprehensions and lambda functions to a reasonable extent
- Use `isinstance` to check for a type
- Use packing & unpacking for multiple values

Best Practices (continued)

- Use logging instead of print
- Avoid using `import *`
- Use `if` instead of `if Bool` or `if len()`
- Avoid the range length idiom, use `for in` and `enumerate` instead (you may need `zip`)
- Think about packaging of your code (modules, functions, paths, etc.)

A Short Excursion Into Software Design

- It’s all about complexity
- Complexity is incremental
- Tactical vs. strategic programming
- Working code is not enough
- Requires investment mindset: continuous, incremental improvements

Thanks!

Any questions ?