EPFL, CIVIL-127

Programming and software development for engineers

How’s final project coming along?

Estimated timeline for your final project

- By the end of this week (4.5.2025)
  - Your agent should be picking up passengers and dropping them off – your algorithm should be able to score points in a room with just itself

Estimated timeline for your final project

- By the end of this week (4.5.2025)
  - Your agent should be picking up passengers and dropping them off – your algorithm should be able to score points in a room with just itself
- By the end of next week (11.5.2025)
  - Your agent should be beating at least one of the staff's bots – it's likely that your algorithm is as good or better than a human player with a keyboard

Estimated timeline for your final project

- By the end of this week (4.5.2025)
  - Your agent should be picking up passengers and dropping them off – your algorithm should be able to score points in a room with just itself
- By the end of next week (11.5.2025)
  - Your agent should be beating at least one of the staff's bots – it's likely that your algorithm is as good or better than a human player with a keyboard
- Following week (18.5.2025)
  - Start cleaning up your code – make sure you have comments, tests, etc. – think about what you are going to put in your one pager

Estimated timeline for your final project

- By the end of this week (4.5.2025)
  - Your agent should be picking up passengers and dropping them off – your algorithm should be able to score points in a room with just itself
- By the end of next week (11.5.2025)
  - Your agent should be beating at least one of the staff's bots – it's likely that your algorithm is as good or better than a human player with a keyboard
- Following week (18.5.2025)
  - Start cleaning up your code – make sure you have comments, tests, etc. – think about what you are going to put in your one pager
- Submission deadline: May 23rd, 2025 @ 6pm

Type Hints

Types are **contracts** between callees and callers

Types are contracts between callees and callers

```python
def f1(a: int) -> int:
    return a * 2


def f2():
    print(f1(123))
```

- Contract == agreement
- caller == code calling a function or method
- callee == function or method being called
- Example
  - `f1` will behave correctly when called with an integer. It'll return an integer.
  - The last line honors the contract, it calls `f1` with an integer.

One time setup

- `pip install mypy`
- Install mypy VSCode extension
- Configure `python.analysis.typeCheckingMode` in your project's .vscode/settings.json file
  - off → no type checking
  - strict → type hints needed for all code
  - basic / standard → partial use of type hints
- note: mypy has many configuration options, the tool's behavior can therefore vary from project to project / use-case to use-case

Documentation

- https://docs.python.org/3.12/library/typing.html (Support for type hints)
  - Type hints cheat sheet
  - Type System Reference
  - Static Typing with Python
  - Specification for the Python type system

Pros of using type hints

- Better code clarity
  - Types serve as documentation.
  - E.g. `-> int` is less characters than `"""returns an integer"""`
- Forces better code structure
- Early bug detection
  - Type-hints are well structured and can be processed by tools (such as a type checker)
  - Detect bugs in your code automatically and without running the code (static analysis)

Pros of using type hints

- Improves IDE experience
  - Go to a type definition
  - Find where a type is used
  - Autocompletion
- Can potentially result in faster code
  - The Python runtime can potentially use type information to generate more optimal code
- Optional
  - Type hints can be used only where desired

Cons of using type hints

- Learning curve
  - Type hints have been part of Python since 2015, but some people aren't yet familiar with them
  - Python 3.12 doesn't retain or enforce the types when the code runs
- It's a retrofit, with some drawbacks
  - Types are dropped at runtime, they aren't enforced when the code runs
  - Many minor tradeoffs
- Optional
  - Can lead to false sense of safety when a codebase is partially typed

Type hints make the code easier to read

```
def center(x1, y1, x2, y2):
    """
    Calculates the center of a rectangle
    x1, y1 are the first corner (integer values).
    x2, y2 are the opposite corner (integer values).

    Returns a tuple of integers
    """
    dx = (x2 - x1) / 2
    dy = (y2 - y1) / 2
    return (x1 + dx, y1 + dy)
```

Type hints make the code easier to read

```python
def center(x1: int, y1: int, x2: int, y2: int) -> tuple[float, float]:
    """
    Returns the center of a rectangle given two
    opposite corners.
    """
    dx = (x2 - x1) / 2
    dy = (y2 - y1) / 2
    return (x1 + dx, y1 + dy)
```

Detect bugs earlier

```python
def center(x1: int, y1: int, x2: int, y2: int) -> tuple[int, int]:
    """
    Returns the center of a rectangle given two opposite corners.
    """
    dx = (x2 - x1) / 2
    dy = (y2 - y1) / 2
    return (x1 + dx, y1 + dy)
```

type_hints.py 1 of 2 problems

```
Type "tuple[float, float]" is not assignable to return type "tuple[int, int]"
  "float" is not assignable to "int"
  "float" is not assignable to "int" Pylance(reportReturnType)
```

1998 Mars Climate Orbiter

- $300 million NASA robotic space probe
- Software mixed meters (metric) and feets (imperial) during a critical calculation resulting in permanent loss of the spacecraft

Open question: would some software development techniques have caught this bug during the development phase?

Improved IDE experience

```python
class Stack:
    def __init__(self):
        self.s: list[int] = []

    def push(self, x: int):
        self.s.append(x)

    def pop(self) -> int:
        return self.s.pop()

    def max(self) -> int:
        return max(self.s)
```

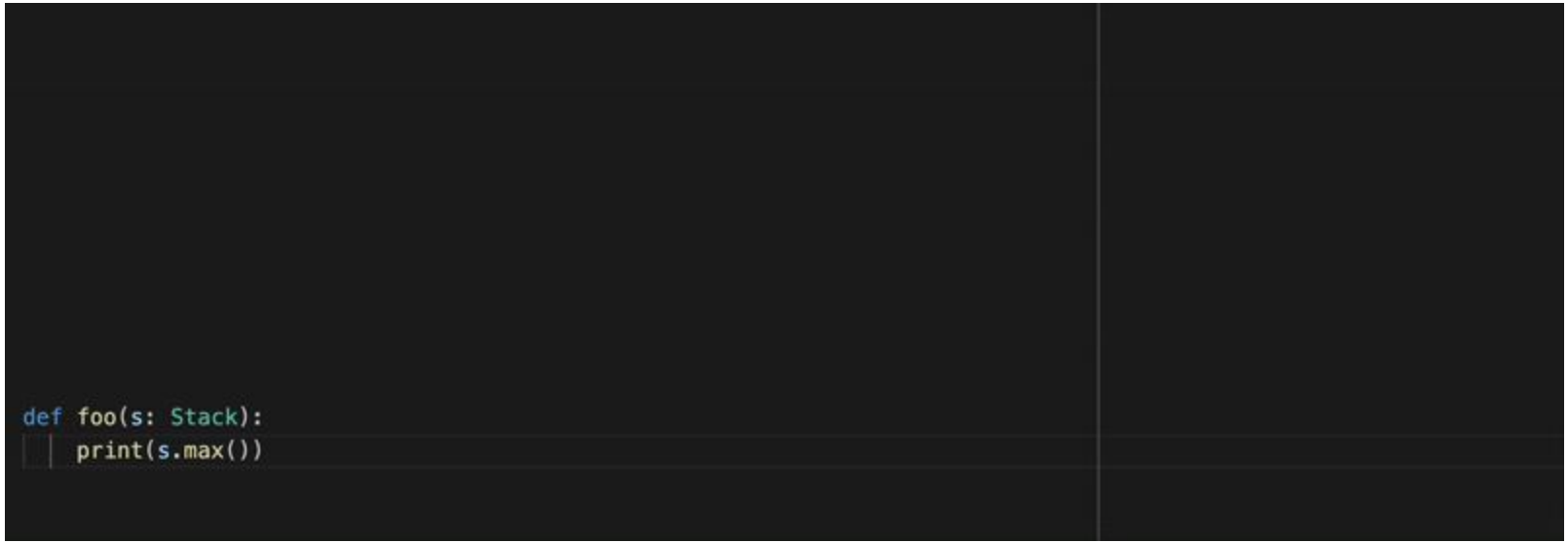
- We annotate our `Stack` class from a few weeks ago

Improved IDE experience

```
def foo(s: Stack):
    print(s.max())
```

- We annotate our variable `s`, to be of type Stack
- Typing `s.` then lists all the available methods and their arguments
- Tools such as Copilot can potentially use the type hints to provide better suggestions

Improved IDE experience



Where can type hints appear?

- Function and method parameters
  - `def foobar(x: str):`
- Function and method return type
  - `def foobar(x) -> int:`
- Function and method parameters + return type
  - `def foobar(x: str) -> int:`

Where can type hints appear?

```
class Stack:
    def __init__(self) -> None:
        self.s: list[int] = []
```

- Instance variables can be assigned types in the `__init__` method

Where can type hints appear?

```python
import random


def foobar(x: int) -> None:
    print(x)


x = []
while True:
    x.append(random.randint(0, 9))
    if sum(x) == 100:
        break
foobar(len(x))

```

- Local variable have their types inferred
- In some rare cases, you need to help the type inference system

Where can type hints appear?

```
import random


def foobar(x: int) -> None:
    print(x)


x: list[int] = []
while True:
    x.append(random.randint(0, 9))
    if sum(x) == 100:
        break
foobar(len(x))
```

- Local variable have their types inferred
- In some cases, the type cannot be inferred
- `x: list[int] = []` specifies that `x` has type `list[int]`

Where can type hints appear?

```
import random


def foobar(x: int) -> None:
    print(x)


x = list[int]()
while True:
    x.append(random.randint(0, 9))
    if sum(x) == 100:
        break
foobar(len(x))
```

- Local variable have their types inferred
- In some cases, the type cannot be inferred
- `x = list[int]()` is equivalent

Type derived from a class

```python
class Stack:
    def __init__(self) -> None:
        self.s: list[int] = []

    def push(self, x: int) -> None:
        self.s.append(x)

    def pop(self) -> int:
        return self.s.pop()

    def max(self) -> int:
        return max(self.s)

def foobar(x: Stack) -> None:
    print(x)

foobar(Stack())
```

- Every class defines a type with the same name
- `class Stack` defines a `Stack` type
- Instance attributes and their types are derived from the body of `__init__`

`bool` type

```
def foobar(x: bool) -> None:
    print(x)

foobar(True)
foobar(False)
```

- `True` and `False` have type `bool`

`int` type

```
def foobar(x: int) -> None:
    print(x)

foobar(0)
foobar(123)
```

- `0` and `123` have type `int`

`float` type

```
def foobar(x: float) -> None:
    print(x)

foobar(4.56)
```

- `4.56` has type `float`

`int` subtype of `float`

```
def foobar(x: float) -> None:
    print(x)


foobar(40)
```

- `int` is a subtype of `float`, you can use an `int` whenever you need a `float`

`str` type

```
def foobar(x: str) -> None:
    print(x)

foobar("hello")
```

- `"hello"` has type `str`

`tuple` type

```
def foobar(x: tuple[int, str]) -> None:
   print(x)

foobar((1, "x"))
```

- `(1, "x")` has type `tuple[int, str]`

union type

```
def foobar(x: int | str) -> None:
    print(x)


foobar(1)
foobar("x")
```

- `type1 | type2 | type3 | …` defines a union type
- A union type is a type which matches either of several choices

union type

```
from typing import Union

def foobar(x: Union[int, str]) -> None:
   print(x)

foobar(1)
foobar("x")
```

- `Union[type1, type2, type3]` is an equivalent syntax
- `Union[type1, Union[type2, type3]]` would also be equivalent
- `|` is more readable (?)

`list` type

```
def foobar(x: list[int]) -> None:
    print(x)


foobar([1, 2, 3])
```

- Homogenous lists can be typed with `list[type]`
- `[1, 2, 3]` has type `list[int]`

`list` type

```python
def foobar(x: list[int | str]) -> None:
    print(x)

foobar([1, "foo", 3])
```

- Non-homogenous lists can be typed using union types
- Such non-homogenous lists are code smells: something is poorly designed

`set` type

```python
def foobar(x: set[str]) -> None:
    print(x)



foobar({"e", "h", "l", "o"})
```

- `{"e", "h", "l", "o"}` has type `set[str]`
- As with lists:
  - Use a union type for non-homogenous sets
  - Non-homogenous sets are code smells

`dict` type

```
def foobar(x: dict[str, int]) -> None:
   print(x)



foobar({"a": 1, "b": 10})
```

- `{"a": 1, "b": 10}` has type `dict[str, int]`
- This type is usually not enough
  - You usually want to describe the dictionary's shape – which keys are required and the associated types for their values

TypedDict

```python
from typing import NotRequired, TypedDict

class Star(TypedDict):
    pos: tuple[int, int]
    name: NotRequired[str]

def foobar(x: Star) -> None:
    print(x)

foobar({"pos": (10, 15)})
foobar({"pos": (10, 15), "name": "alpha"})
```

- A class which extends `TypedDict` becomes a regular dict at runtime
- Attributes are used to define the shape of the dictionary
- `NotRequired` indicates optional keys

Enum

```python
from enum import StrEnum


class Color(StrEnum):
    Red = "red"
    Blue = "blue"


def foobar(x: Color) -> None:
    print(x.value)


foobar(Color.Red)   # prints red
```

- Use `Enum` if you don't care about the value's type
- Use `StrEnum` if the values are strings
- Use `IntEnum` if the values are integers

Remember, most type hints don’t exist at runtime

```python
def foobar(x: list[int]) -> None:
   print(type(x))

foobar([1, 2, 3])  # prints <class 'list'>
```

- At runtime
  - Python keeps track of primitive types and classes
  - The exact type hints are lost

Any

```python
from typing import Any


def foobar(x: Any) -> None:
    print(x)


foobar("foobar")
foobar(123)
```

- `Any` is a type which matches everything
- Almost equivalent to not using type hints
- Almost equivalent to using `object`, the base class for standard types and all classes

Optional

```python
from typing import Optional

def foobar(x: Optional[str]) -> None:
   print(x)

foobar(None)  # prints None
foobar("abc")  # prints abc
```

- `Optional[type]` is equivalent to `type | None`

Alias

```
type Foo = list[tuple[str, int]]


def foobar(x: Foo) -> None:
    print(x)


foobar([("a", 1), ("b", 10)])
```

- `type Foo = …` creates an alias
- Makes the code less verbose and more readable

NewType

```python
from typing import NewType

Sciper = NewType('Sciper', int)

def foobar(x: Sciper) -> None:
    print(x)

foobar(Sciper(123))
```

- `foo = NewType(...)` creates a new type
- A way to “mask” the underlying type
- `Sciper` behaves like an `int`, but you can’t accidentally use some variable which happens to also be an `int`
- Note: “Sciper” is repeated twice, once as the first parameter for `NewType` and once as the actual type.

Callable

```python
from typing import Callable

def foobar(x: Callable[[str], int]) -> None:
    print(x("foobar"))

foobar(len)  # prints 6
foobar(lambda x: x.count("o"))  # prints 2
```

- You'll rarely type hint a callable (we already discussed using classes reduces the need to have functions or lambdas as parameters!)
- `Callable[[str], int]` is the type of a function or lambda which takes a `str` and returns an `int`

Generic classes

```python
class Stack[T]:
    def __init__(self) -> None:
        self.s: list[T] = []

    def push(self, x: T):
        self.s.append(x)

    def pop(self) -> T:
        return self.s.pop()
```

- `[T]` can be used to parameterize a class
- `T` will get replaced by an actual type when the class is instantiated
- You can use any identifier instead of `T`. Typically, single letter and capitalized (e.g. `T`, `U`, `K`, and `V`. `T1`, `T2`, ... is also common)
- You will probably not write generic classes, but you will read code which uses generics

Generic classes

```
def foobar() -> None:
    s = Stack[int]()
    s.push(1)
    s.push(2)
    print(s.pop())   # prints 2

foobar()
```

- `s = Stack[int]()` causes the replacement of `T` with `int`

Generic classes

```
def foobar() -> None:
    s: Stack[int] = Stack()
    s.push(1)
    s.push(2)
    print(s.pop())   # prints 2

foobar()
```

- `s: Stack[int] = Stack()` is equivalent

match

```python
def f1(x: str) -> None:
    print(x, x)


def f2(x: int) -> None:
    print(x * 2)


def foobar(x: str | int) -> None:
    f1(x)
```

type_hints.py 1 of 2 problems

```
Argument of type "str | int" cannot be assigned to parameter "x" of type "str" in function "f1"
  Type "str | int" is not assignable to type "str"
    "int" is not assignable to "str" Pylance(reportArgumentType)
```

match

```
def f1(x: str) -> None:
   print(x, x)

def f2(x: int) -> None:
   print(x * 2)

def foobar(x: str | int) -> None:
   match x:
       case str():
           f1(x)
       case int():
           f2(x)

foobar("abc")  # prints abc abc
foobar(12)  # prints 24
```

- `match` destructures union types
- Within each case, the type of `x` becomes more specific
  - Inside `case str():` the type for `x` becomes str
  - Inside `case int():` the type for `x` becomes int
- Only works for types which are preserved at runtime

`match`

- `match` combined with `assert_never`, can be used to statically enforce that every permutation for a type has been handled
- Most useful when combined with enums
- In the next example, we forgot a case statement for `Color.Blue`

match

```python
from enum import Enum
from typing import assert_never

class Color(Enum):
    Red = "red"
    Black = "black"
    Blue = "blue"

def foobar(x: Color) -> None:
    match x:
        case Color.Red:
            pass
        case Color.Black:
            pass
        case _:
            assert_never(x)
```

type_hints.py 1 of 2 problems

```
Argument of type "Literal[Color.Blue]" cannot be assigned to parameter "arg" of type "Never" in function "assert_never"
  Type "Literal[Color.Blue]" is not assignable to type "Never" Pylance(reportArgumentType)
```

Nominal vs structural typing

- Usually, a type system is either nominal or structural
- Nominal == two classes which have the same methods are considered different types
- Structural == two classes which have the same methods are considered the same type
- Python is mostly nominal, except for some convenient cases when it becomes structural

Nominal vs structural typing

```python
class Foo:
    def f(self) -> int:
        return 1


class Bar:
    def f(self) -> int:
        return 2


def foobar(x: Foo) -> None:
    print(x.f())

foobar(Bar())
```

type_hints.py 1 of 2 problems

```
Argument of type "Bar" cannot be assigned to parameter "x" of type "Foo" in function "foobar"
  "Bar" is not assignable to "Foo" Pylance(reportArgumentType)
```

Nominal vs structural typing

```python
from typing import Protocol

class Foo(Protocol):
    def f(self) -> int:
        return 1

class Bar:
    def f(self) -> int:
        return 2

def foobar(x: Foo) -> None:
    print(x.f())

foobar(Bar())  # prints 2
```

- Extending `Protocol` enables a class to use structural typing