Type Hints

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Code without type hints

```
class Student:
    def __init__(self, name, student_id, tuition_balance):
        self.name = name
        self.student_id = student_id
        self.tuition_balance = tuition_balance

# Trying to create a class can be tricky
walker = Student("Sarah Walker", 319921, 15000)
```

- Is `student_id` supposed to be an integer?
- Should `tuition_balance` be a float or an integer? Maybe a string?
- After `walker` has been created, how can we remember what type it is later in code?

Type hints

Optional tool that allows for information about an object to be added to code

- Easier to read, troubleshoot
- One of the **best ways to demonstrate enterprise-grade Python skills**
- Not enforced by the interpreter
- Built-in type keywords, `typing` library, custom classes

```
# Add type hinting when creating a
# variable
gpa: float = 3.92
```

```
# Add type hinting to a function/method
# definition
def check_grades(year: str) -> List[int]:
    ...
```

```
# Can even use custom classes
students: Dict[str, Student] = {...}
```

Type hinting with built-in type keywords

```
# Type hinting for declarative logic
name: str = "Frost"  # Before: name = "Frost"
student_id: int = 91031367
tuition_balance: float = 17452.78
```

```
# Type hinting for function/method definitions
def get_schedule(semester: str) -> dict:
    ...
```

- Use the syntax `variable: type`
- Declarative, signature of function/method
- `def .... () -> type:` to specify return type

typing Library

`typing` is a library used to provide more tools to type hint

- `List`, `Dict`, `Tuple`
- Hint top-level objects, and elements of objects

```
from typing import List, Dict

student_names: List[str] = ["Morgan", "Chuck", "Anna"]
student_gpas: Dict[str, float] = {
    "Casey": 3.71,
    "Sarah": 4.0
}
```

- `Any`, `Set`, `Iterator`, `Callable`

Type hinting with custom classes

```
class Student:
    def __init__(self, name: str, student_id: int, tuition_balance: float) -> None:
        self.name: str = name
        self.student_id: int = student_id
        self.tuition_balance: float = tuition_balance

    def get_course(self, course_id: str) -> Course:
        ...
        return course
```

```
# Use Student and Course to type hint
walker: Student = Student("Sarah Walker", 319921, 15000)
data_science: Course = walker.get_course("TDM-20100")
```

Checking object types

```
# walker: Student = Student("Sarah Walker", 319921, 15000)
print(type(walker))
```

```
<class '__main__.Student'>
```

```
# data_science: Course = walker.get_course("TDM-20100")
print(type(data_science))
```

```
<class '__main__.Course'>
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Descriptors

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Changing interactions with attributes

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

```
# Create a student object, and access the "ssn" attribute
shaw = Student("Daniel Shaw", "193-80-1821")
print(shaw.ssn)

# What if the value is changed or deleted? How should this be handled?
```

```
193-80-1821
```

Creating descriptors with @property

Descriptors are objects used to regulate the way that an attribute is retrieved, set, or deleted

Create descriptors with `@property`

- "getter", "setter", "deleter"
- Method names == name of attribute.

Descriptors may also be created with...

- `property()` function
- `__get__()`, `__set__()`, `__delete__()`

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

    @property
    def ssn(self):
        return "XXX-XX-" + self._ssn[-4:]

    @ssn.setter
    def ssn(self, new_ssn):
        self._ssn = new_ssn

    @ssn.deleter
    def ssn(self):
        raise AttributeError("Can't delete SSN")
```

@property

```
class Student:
    def __init__(self, name, ssn):
        self.name = name
        self.ssn = ssn  # Does not need to include an underscore

    @property
    def ssn(self):  # This is the "getter" method
        return "XXX-XX-" + self._ssn[-4:]
```

- Controls how `ssn` is retrieved
- Interact with `self._ssn`, not `self.ssn`

@ssn.setter

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

    @ssn.setter
    def ssn(self, new_ssn):
        if len(new_ssn) == 11:
          # Add things such as data
          # validation, operations on
          # other attributes, etc.
          self._ssn = new_ssn
```

`@<attribute-name>.setter`

- "setter" method for `ssn`
- Validate data quality
- Perform operations on other attributes

Remember to interact with `self._ssn` , not `self.ssn` !

@ssn.deleter

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

    ...

    @ssn.deleter
    def ssn(self):
        # Can perform clean up, soft delete, raise exception
        raise AttributeError("Can't delete SSN")
```

Descriptors in action

```
shaw = Student("Daniel Shaw", "193-80-1821")
print(shaw.ssn)  # Access the ssn attribute
```

```
XXX-XX-1821
```

```
shaw.ssn = "821-11-9380"  # Update Shaw's social security number
print(shaw.ssn)
```

```
XXX-XX-9380
```

```
del shaw.ssn  # Attempt to delete the ssn attribute
```

```
AttributeError: Can't delete SSN
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Customizing Attribute Access

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

AttributeError

```
class Student:
  def __init__(self, student_name, major):
    self.student_name = student_name
    self.major = major
...
```

```
karina = Student("Karina", "Literature")
student.residence_hall  # Attempt to access an attribute that does not exist
```

```
...
AttributeError: 'Student' object has no attribute 'residence_hall'
```

__getattr__()

```
# Rest of the class definition above

  def __getattr__(self, name):
    # Implement logic here
    ...
```

An object's namespace is a collection of attributes that are associated with that object

`__getattr__()` is executed when an attempt to reference **ANY attribute** outside of an object's namespace** is made

- Magic method, not called directly
- Takes a `name` parameter
- Implements custom functionality, rather than raising an `AttributeError`

Resolving the AttributeError

```
class Student:
  def __init__(self, student_name, major):
    self.student_name = student_name
    self.major = major

  def __getattr__(self, name):
      print(f"""{name} does not exist in this object's namespace, try setting
            a value for {name} first""")
```

```
karina.residence_hall  # Now, try to retrieve the residence_hall attribute again
```

```
residence_hall does not exist in this object's namespace, try setting a value for
  residence_hall first
```

__setattr__()

`__setattr__()` is a magic method that is executed when a (new or existing) attribute is **set** or **updated**

- Includes attributes set using `__init__()`
- Takes `name` of attribute and `value`
- Leverages `__dict__` attribute of the object

```
# Rest of the class definition above

  def __setattr__(self, name, value):
    # Implement logic here
    ...

    # Use __dict__ to create/update
    # the attribute
    self.__dict__[name] = value
```

Controlling changes to attributes, validation, transformation

`__dict__` stores all attributes of the object, can be used to retrieve and store data

Customizing attribute storage

```
class Student:
  def __init__(self, student_name, major):
    self.student_name = student_name
    self.major = major

  def __setattr__(self, name, value):
    # If value is a string, set the attribute using the __dict__ attribute
    if isinstance(value, str):
      print(f"Setting {name} = {value}")
      self.__dict__[name] = value

    else: # Otherwise, raise an exception noting an incorrect data type
        raise Exception("Unexpected data type!")
```

__setattr__ in action

```
# Set an attribute using a value of type 'str'
karina.residence_hall = "Honors College South"
print(karina.residence_hall)
```

```
Setting residence_hall = Honors College South
Honors College South
```

```
# Set an attribute using a value of type 'int'
karina.student_id = 19301872
```

```
...
    raise Exception("Unexpected data type!")
Exception: Unexpected data type!
```

Using __getattr__ and __setattr__ together

```
class Student:
    ...

    def __getattr__(self, name):
        # Set the attribute with a placeholder
        self.__setattr__(name, None)
        return None

    def __setattr__(self, name, value):
        if value is None:  # Print a message denoting a placeholder
            print(f"Setting placeholder for {name}")

        self.__dict__[name] = value  # Set the attribute
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Custom Iterators

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Iterators

Classes that allow for a collection of objects or data stream to be traversed, and **return one item at a time**

- Similar to `list` 's, `tuple` 's, but act differently
- Navigate, transform, generate
- Looped over using `for` loops
- `next()` function

Iterator protocol...

```
# Collection
chuck = NameIterator("Charles Carmicheal")
for letter in chuck:
    print(letter)
```

```
C
h
u
...
```

```
# Data stream
fun_game = DiceGame(rolls=3)
next(fun_game)  # ... and so on
```

```
4
```

Iterator protocol

`__iter__()`

- Returns an iterator, in this case, a reference to itself
- `... return self`

`__next__()`

- Returns the **next** value in the collection or data stream
- Iteration, transformation, and generation takes place

Both `__iter__()` and `__next__()` must be defined for a class to be considered an iterator!

Example iterator

```
class CoinFlips:
    def __init__(self, number_of_flips):
        self.number_of_flips = number_of_flips  # Store the total number of flips
        self.counter = 0

    def __iter__(self):
        return self  # Return a reference of the iterator

    # Flip the next coin, return the output
    def __next__(self):
        if self.counter < self.number_of_flips:
            self.counter += 1
            return random.choice(["H", "T"])
```

Using an example iterator

```
three_flips = CoinFlips(3)


# Flip the coin three times
next(three_flips)
next(three_flips)
next(three_flips)
```

```
H
H
T
```

Looping through an iterator

```
three_flips = CoinFlips(3)

# Now, try to loop through every element of the iterator
for flip in three_flips:
    print(flip)
```

```
T
H
T
None
None
None
...
```

StopIteration

```
...
    def __next__(self):
        # Only do this if the coin hasn't been flipped "number_of_flips" times
        if self.counter < self.number_of_flips:
            self.counter += 1
            return random.choice(["H", "T"])

        else:  # Otherwise, stop execution with StopIteration
            raise StopIteration
```

- Signals end of collections/data stream
- Prevents infinite loops
- Easy to handle

Looping through an iterator

```
three_flips = CoinFlips(3)


# Now, try to loop through every element of the iterator
for flip in three_flips:
    print(flip)
```

```
H
T
H
```

Handling StopIteration exceptions

```
while True:
    try:
        next(three_flips)  # Pull the next element of three_flips

    # Catch a stop iteration exception
    except StopIteration:
        print("Completed all coin flips!")
        break
```

```
H
H
H
Completed all coin flips!
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON