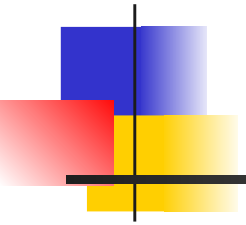


Type Checking

Type Checking

Verify that the rules for using different types are obeyed, and that the correct types are used in function calls, assignments, and other program elements.

Examples:

```
lst = ["cat", "dog", "rat"]
sum( lst )                      # type error

for x in range(1.0,4.0): # type error:
    print(x)                    # int required
```

Static

static - *fixed, unchanging, immobile*

In computer programming:

anything that is done or known before run-time.

"static content" - fixed content in a web application, such as images, fonts, CSS files, fixed web pages.

"**static type checking**" - type checking done before the program is run.

- done by a compiler or static type checking tool.

Dynamic

dynamic - *characterized by change or activity*

In computer programming:

anything that is done, created, or known only when the code is run.

"dynamic content" - web pages generated at run-time from a template. Content that changes over time.

"**dynamic type checking**" - verify type rules while the program is running.

Does Python Do Dynamic Type Checking?

Answer is not obvious.

Consider this:

```
# what type is required for x and y?
def join(x, y):
    return x + y
# join accepts many different types
join(2, 3)
join("hi", "bye")
join(Fraction(1,2), Fraction(2,3))
# but this fails
join(2, "hi")
```

What Some People Say

Python does dynamic type checking.

Python associates types with *values* rather than *variables*.

Type checking is done on values.

Static versus Dynamic Binding

"**Binding**" refers to association of names with particular pieces of code.

Static Binding - a name is "bound" to particular code in an unchanging (static) way.

Dynamic Binding - a name is "bound" to code in a dynamic, changing way (at run-time).

@staticmethod

```
class Fraction:
    @staticmethod
    def gcd( m, n):
        """greatest common divisor"""
        # use Euclid's algorithm
```

gcd is statically bound. We know exactly what code will be invoked even before the program is run!

```
x = Fraction.gcd(60, 75)
```

Dynamic binding

```
lst = [ Fraction(2,3), "hello",
        datetime.now() ]
for x in lst:
    print( str(x) )
```

```
2/3
hello
2019-11-17 15:50:34
```

`str(x)` is dynamically bound to the __str__() method of a particular class (Fraction, string, datetime).

We don't know until run-time what kind of object x refers to, or which class's __str__() method should be invoked.

Dynamic Binding and Polymorphism

Dynamic binding is needed to enable polymorphism.

The example from previous slide uses polymorphism.

```
lst = [ Fraction(2,3), "hello",
        datetime.now() ]
for x in lst:
    print( str(x) )
2/3                 -  __str__ of Float
hello               -  __str__ of string
2019-11-17 15:50:34
                    -  __str__ of datetime
```

Static Checking & Software Correctness

We want our software to be correct (of course).

Static type checking finds some programming errors before the program is run.

Some type errors may also indicate *logic errors*.

Simple Static Type Checking

Specify that "join" only accepts string parameters:

```
def join( x: str, y: str ) -> str:
    return x + y

if __name__ == '__main__':
    a = 2
    b = "hello"
    print( join(a,b) )
```

"mypy" is a static type checking tool. Run it:

```
cmd> mypy join.py
Line 7: error: Argument 1 to "join" has
incompatible type "int"; expected "str"
```

Example

```
class Scorecard:
    """Accumulate scores and compute their average."""
    def __init__(self):
        self.scores = []

    def add_score(self, score):
        self.scores.append(score)

    def average(self):
        """return average of all scores"""
        return sum(self.scores)/max(1,len(self.scores))

if __name__ == "__main__":
    scores = Scorecard()
    n = input("input a score: ")
    scores.add_score(n)
    n = input("input another score: ")
    scores.add_score(n)
    print("The average is " + scores.average())
```

This code contains 2 distinct errors. As written, most IDE won't detect them.

Exercise

1. Download scorecard.py to an empty directory.
2. Open in your favorite IDE.
3. Does the IDE flag any errors?
4. Add *type hints* -- **one at a time** so you can see the effect.
   a) "hint" the parameter to add_score. What happens?
   b) "hint" the scores attribute

```
self.scores: List[float] = []
```

   What happens? Does IDE detect an error in code?
   c) "hint" the return type of `average(self)`.

For VS Code, if you don't have Pyright extension, then install it. See any difference?

Tools for Static Type Checking

1. mypy - https://mypy.readthedocs.io/
   - installation: `pip install mypy`
   - check a file: `mypy filename.py`
   - strict checking: `mypy --strict filename.py`
   - Getting Started Guide has many examples: https://mypy.readthedocs.io/en/latest/getting_started.html
2. PyCharm has built-in static type checking
3. VS Code - add **Pyright** extension for static type checking

Typing and Encapsulation

In Scorecard, the scores are assumed to be numbers.

Can we allow scores to be objects?

score = Score("Quiz 1", 10.0)

In `Scorecard` we could write:

```
def average(self):
    # add the values of the score objects
    total = sum(float(x) for x in self.scores)
    # don't divide by zero if no scores
    return total/max(1, len(self.scores))
```

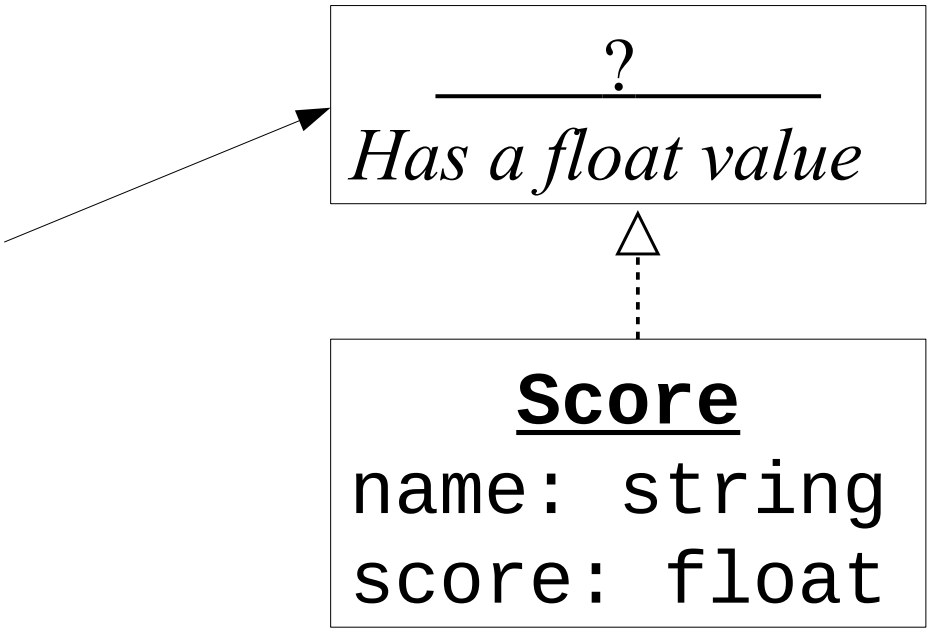
Typing and Encapsulation

What is the ***required behavior*** of a `Score` object,
so that Scorecard can call `float(score)` for any score?

```
def add_score(self, score: ____?____):
```

What "type" specifies: "this object has a float value, and you can call `float(x)` *to get it"?*

See: `typing` *package.*

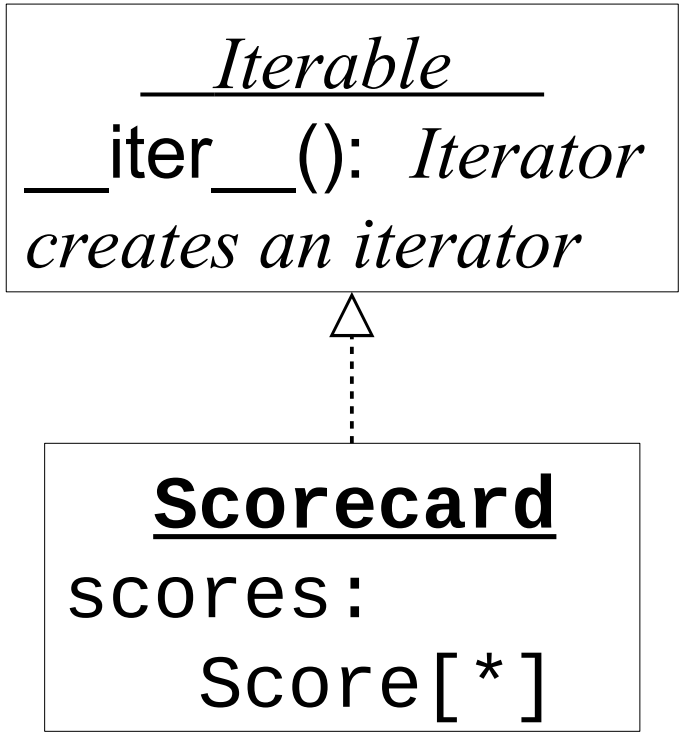


Typing and Behavior

What is the ***required behavior*** of a Scorecard so that we can use Scorecard as data source in a for loop?

```
scorecard = Scorecard()
...  # add some scores

# can this possibly work?
for score in scorecard:
    print(score)
```



for loop

What kind of objects can be used as data in a "for" loop?

```
for x in data:
    print(x)
```

data can be:

string (str)

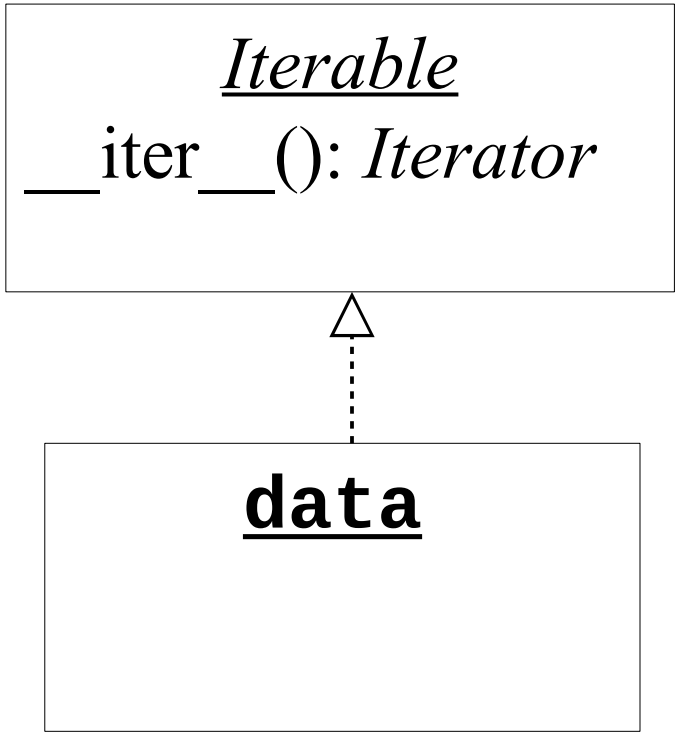
list

dict

range

set

tuple

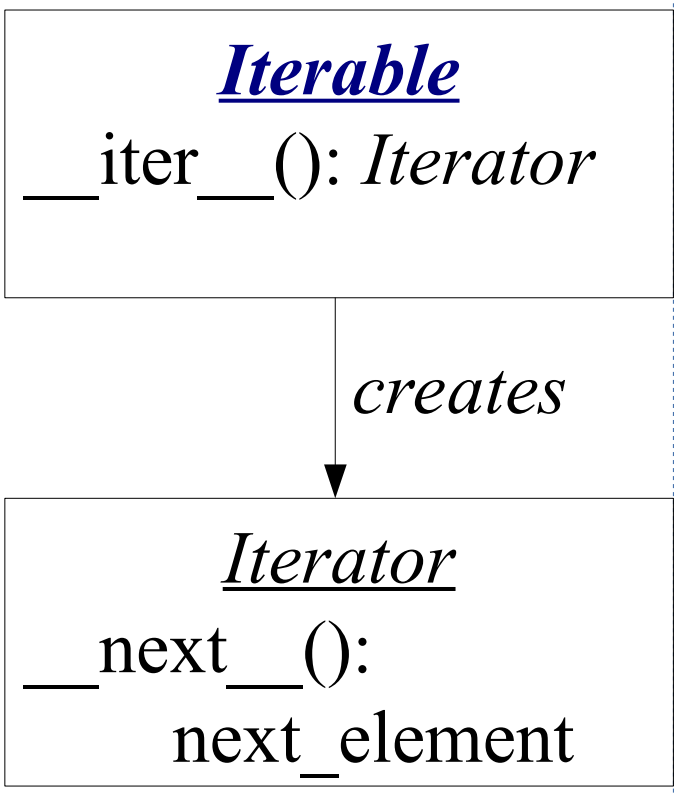


Iterable

Iterable - a type of object (usually a collection) that provides a method for creating an *Iterator.*

Example:

```
# stuff is an Iterable collection
stuff = ("first", "second", "third")
iterator = iter( stuff )
```

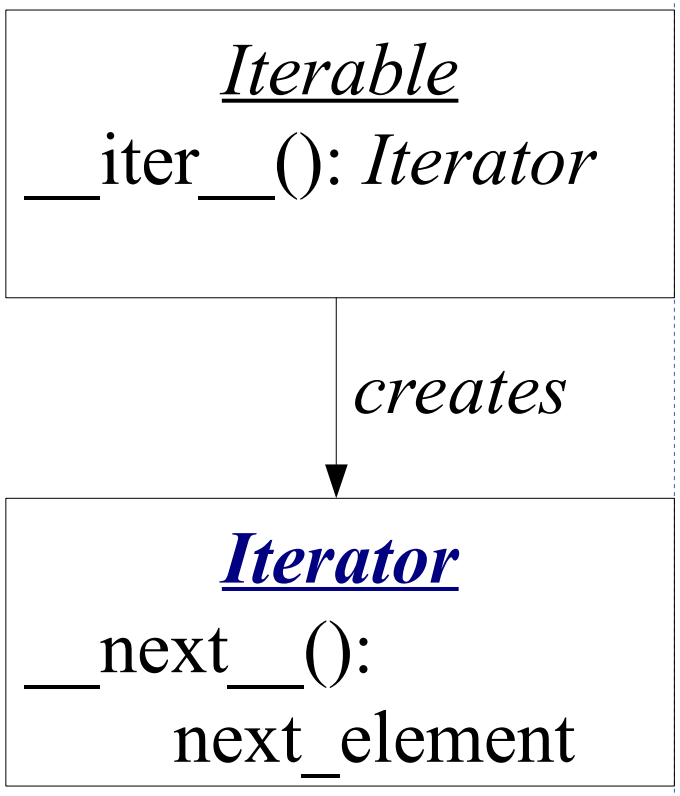


Iterator

Iterator - an object that lets you sequentially (iteratively) access elements from some source. A Python Iterator provides a __next__ method invoked via next(iterator)

Example:

```
# stuff is an Iterable collection
stuff = ("first", "second", "third")
my_iterator = iter( stuff )
# iterate over elements
print( next(my_iterator) )
print( next(my_iterator) )
print( next(my_iterator) )
```



Declare a Class "is" a Type

A Type specifies some behavior (methods).

To declare that your class provides this behavior, write the Type name as a parent type.

Example:

Declare that Scorecard can create an Iterator, and the Iterator returns floats.

```
class Scorecard( Iterable[float] )
    """scorecard creates an iterator for scores"""
   def __iter__(self):
       return iter(self.scores)
```

Types You Should Know

These types specify that a class provides some behavior.

What behavior (methods) does each one guarantee?

`Container`

`Collection`

`Iterable`

`Iterator`

`Dict`

`Mapping`

`List`

`Set`

`Sequence`

Start by reading the collections.abc document page.

Very specific Types

Some types specify a single behavior.

Sized

- can call x.__len__() or len(x)

SupportsFloat

- can call x.__float__() or float(x)

Example:

Declare that Scorecard supports len(scorecard):

```
class Scorecard( Sized )
    def __len__(self):
        """the size is just the number of scores"""
        return len(self.scores)
```

Class Can Provide Many Behaviors

A class can declare that it provides many different kinds of behavior, using types.

Example:

Scorecard creates Iterators and has a length.

```
class Scorecard( Iterable[float], Sized )
    def __len__(self) -> int:
        """the size is just the number of scores"""
        return len(self.scores)

    def __iter__(self) -> Iterator[float]:
        """return an iterator for scores"""
        return iter( self.scores )
```

Resources

Mai's write-up on "type hinting" in ISP19/problems

https://github.com/ISP19/problems/tree/master/type-hints

Python **`typing`** package - defines types

https://docs.python.org/3/library/typing.html

Python abstract base collections (abc) package

https://docs.python.org/3/library/collections.abc.html

This page explains the behavior and methods each collection type provides.

Helps you understand "types" in the typing package.

Another Resource

Mypy Getting Started Guide many short examples of adding type hints to code.

https://mypy.readthedocs.io/en/latest/getting_started.html

Python Type Checking Guide on *RealPython*

https://realpython.com/python-type-checking/

Describes dynamic typing, duck typing, and how to use type hinting.

Iterators

Python Iterators explains difference between Iterable and Iterator, with examples

https://www.w3schools.com/python/python_iterators.asp

Iterators, Generators, Containers, and itertools has more detailed explanation, with code examples.

https://www.datacamp.com/community/tutorials/python-iterator-tutorial