Python Object-Oriented Program with Libraries

Unit 1: Object-Oriented Programming

CHAPTER 5: INTERFACE AND DUCK TYPING

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Python does not have interface. Duck function and Type Inference are used in lieu of interfaces.
- Understand what duck typing is.
- Try to use Python interface and function closure for advanced programming

Duck Typing (Interface)

LECTURE 1

Python duck typing (or automatic interfaces)

- Duck typing is a feature of a type system where the semantics of a class is determined by his ability to respond to some message (method or property).
- The canonical example (and the reason behind the name) is the duck test: If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.
- Duck Typing needs to be implemented by
  - **Type Inference (Type Propagation)**

Polymorphic Method

A method of same name owned by different objects.

```
1   # duck0.py
2   # This example shows a polymorphic method
3   class Duck:
4       def quack(self):
5           print('Quack!')
6   class Goose:
7       def quack(self):
8           print('Quack')
9
10  Goose().quack()     # Goose() create instant temporary object
11  Duck().quack()      # Duck() create instant temporary object
```

```
Run  duck0
     C:\Pytl
     Quack
     Quack!
```

- **Goose()** and **Duck()** create two objects of different types. They shared a same polymorphic method.
- Python does not require type declaration. The object type is assigned at run-time (dynamic type assignment).

Interface (Automatic Interfaces)

A interface is a reference which can represent objects of different types.

Generic Methods

Methods which can be operated on objects of different type. (Or, objects of same interface.)

Java Interface:

Reference of Objects of different classes sharing the same polymorphic methods.

```
interface IEngine {
    void turnOn();
}

public class EngineV1 implements IEngine {
    public void turnOn() {
        // do something here
    }
}

public class Car {
    public Car(IEngine engine) {
       this.engine = engine;
    }

    public void run() {
        this.engine.turnOn();
    }

}
```

Python Class:
No need to declare interface. Just define polymorphic method. (Methods of same name)
This is called **Duck Typing**.

Duck Typing: methods of same name and working similarly.
(Walk like ducks. Swim like ducks.)

```
class Car:
   def __init__(self, engine):
      self.engine = engine

   def run():
       self.engine.turn_on()
```

Duck Typing Example

```
class Duck:
        def quack(self):
                print "Quack, quack!"
        def fly(self):
                print "Flap, Flap!"
class Person:
        def quack(self):
                print "I'm Quackin'!"
        def fly(self):
                print "I'm Flyin'!"
def in_the_forest(thing):
        thing.quack()
        thing.fly()

in_the_forest(Duck())
in_the_forest(Person())
```

Polymorphic Method

Polymorphic Method

Generic Method

Demo Program: duck1.py

Go PyCharm!!!

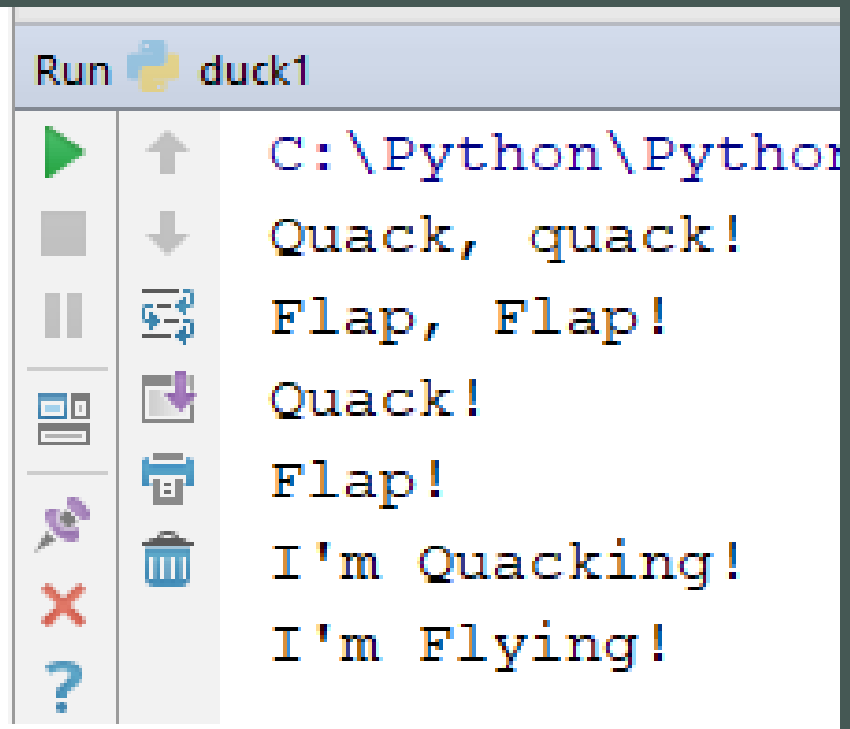
```python
class Duck:
    def quack(self):
        print("Quack, quack!")
    def fly(self):
        print("Flap, Flap!")

class Goose:
    def quack(self):
        print("Quack!")
    def fly(self):
        print("Flap!")

class Person:
    def quack(self):
        print("I'm Quacking!")
    def fly(self):
        print("I'm Flying!")
# Generic Method: A method handle different object types

def in_the_forest(thing):
    thing.quack()      # both quack() and fly() are polymorphic methods
    thing.fly()

in_the_forest(Duck())
in_the_forest(Goose())
in_the_forest(Person())
```

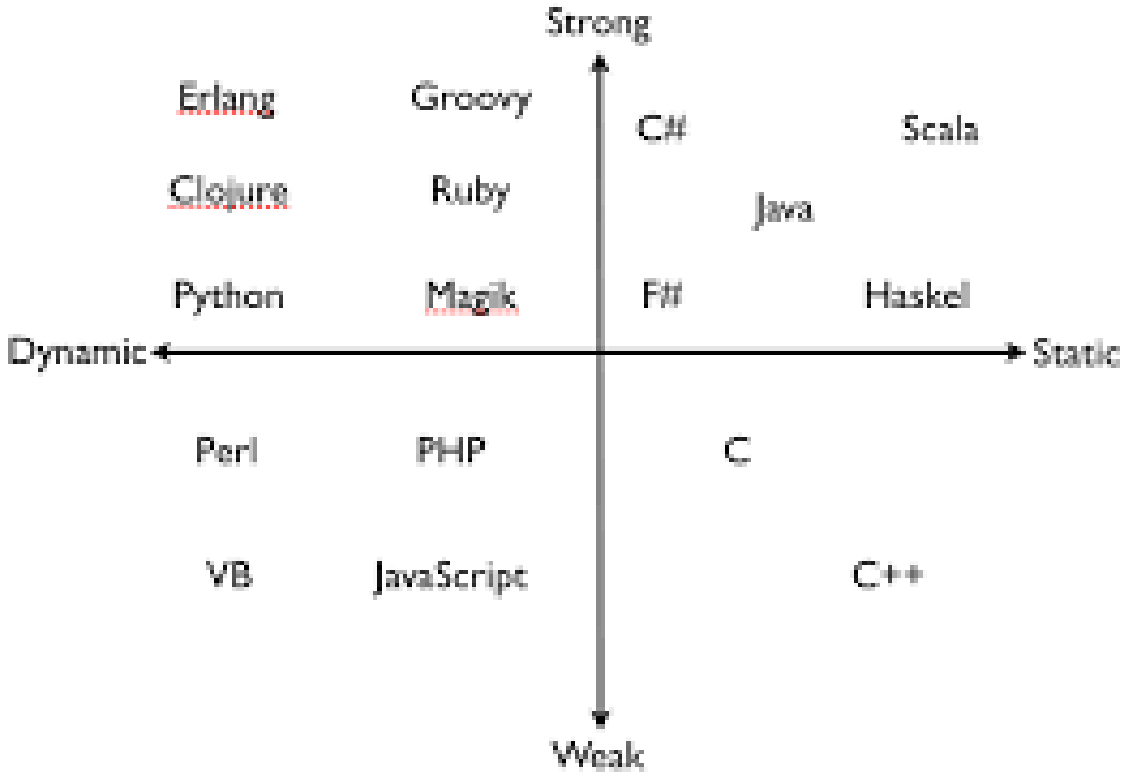


Python Types

LECTURE 2

Python Dynamic Typing

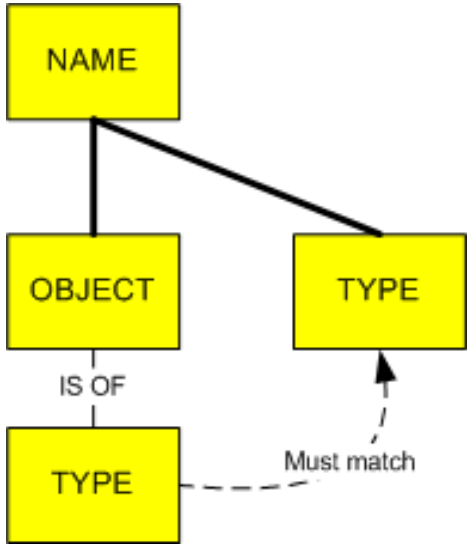
- Python is a dynamically-typed language.
- Java is a statically-typed language.
- Python is a weak-typed (weaker) language.
- Java is a strong-typed (stronger) language.
- Python has type inference. Java does not.



In a **statically typed language**, every variable name is bound both

- to a type (at compile time, by means of a data declaration)
- to an object.

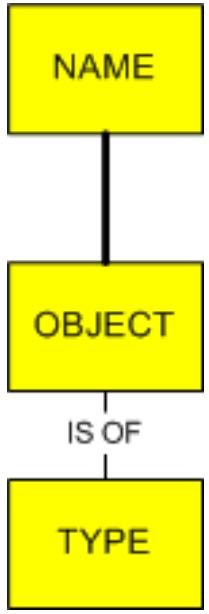
The binding to an object is optional — if a name is not bound to an object, the name is said to be *null.*



Note:
Once a variable name has been bound to a type (that is, declared) it can be bound (via an assignment statement) only to objects of that type; it cannot ever be bound to an object of a different type. An attempt to bind the name to an object of the wrong type will raise a type exception.

In a **dynamically typed language**, every variable name is (unless it is null) bound only to an object.

Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program.



Python Variables can be assigned with Values of Different Types

- We've noticed before that Python is happy to allow us to store any kind of value in any variable we'd like.

```
x = 3
y = 'Boo'
z = [1, 2, 3]
```

- We've also seen that we can potentially change the type of a variable any time we'd like by simply assigning a value of a different type into it.

```
x = (1, 2)      # x is now a tuple
y = 9.5         # y is now a float
z = 'Alex'      # z is now a str
```

Python Types is Associated with Right-Hand-Side Value. Not Left-Hand-Side Reference.

- Or, thought differently, variables themselves don't have types at all in Python; only the values of those variables have types.
- If we use a variable after assigning it a value, what we're allowed to do with it — the operators we can use, the **functions** into which we can pass it as an argument, and so on — is determined by the type of its value at the time we use it.
- Forming interface automatically with polymorphic methods. (Duck Typing) Otherwise, exceptions will be raised.

With Polymorphic Methods

```
w = 'Alex'
print(len(w))   # prints 4
q = 57
print(len(q))   # raises an exception, because ints don't have a length
```

Use of Duck Typing

LECTURE 3

Use of Duck Typing

1. **Declare a global variable** and assign value of different type later.
2. **Polymorphic method**: object behaves similarly.
3. **Generic method** (Forming interface automatically): Generic method can only operate on objects which has polymorphic methods (methods of same name).
4. **Generic Container**: generic data structures (list, tree, graph, matrix, hash table, set, map, associated memory)

How duck typing affects the way we write functions

```
def foo(x, y):
    return x.bar(y) * 2
```

Type Inference

- Let’s left the types out of the function's signature, because it's not as clear what they are until we stop to think about it.
- What must be true about the types of x and y in order to successfully evaluate a call to foo(x, y)?

How duck typing affects the way we write functions

- **x** must be an **object** of some class that has a method called **bar** that takes one parameter (in addition to **self**). There might be many classes like this, and it may not always be the case that all their **bar** methods even do the same thing; the presence of the method is one part of what makes this legal.
- **y** must have a type that is compatible as an **argument** to **bar**. Depending on **x**'s type — and depending on what its **bar** method does — this constraint will be different. Any combination that works is potentially legal.

How duck typing affects the way we write functions

- The type of value returned from the **bar** method must be something that can be multiplied by 2.
- At first blush, that sounds like it must be a number, but if you think harder, you'll remember that you can also multiply other kinds of things (e.g., lists, strings) by numbers, too. (If that sounds weird to you, try evaluating **[1, 2, 3] * 2** in a Python interpreter and see what you get back.)
- See Type Propagation in the next section:
  1. **Object type.**
  2. **Argument**
  3. **Operand type**

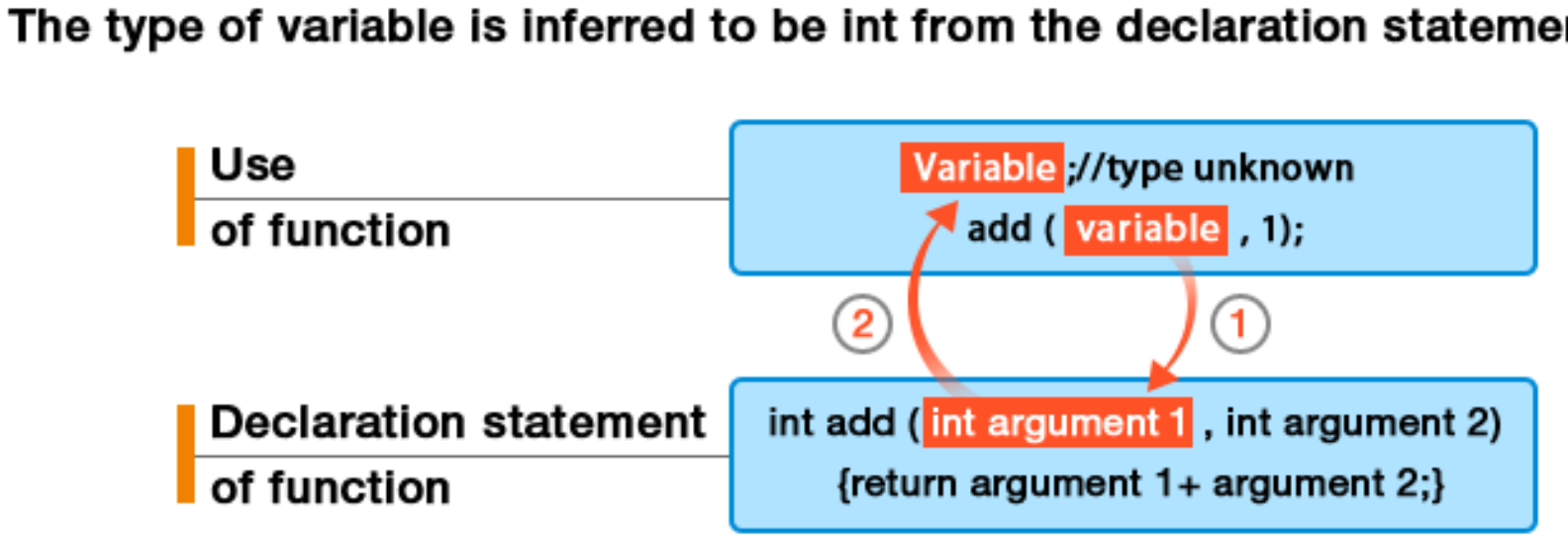
Type Inference

LECTURE 4

Type Inference

- What determine the type of overall expression? (Type Checking for the Output)
- The result of an arithmetic operator usually has the same type as the operands. The result of a comparison is usually Boolean.
- The result of a function call has the type declared in the function's header. The result of an assignment (in languages in which assignments are expressions) has the same type as the left-hand side. In a few cases, however, the answer is not obvious. In particular, operations on subranges and on composite objects do not necessarily preserve the types of the operands.

Type Inference Permeates Modern Languages

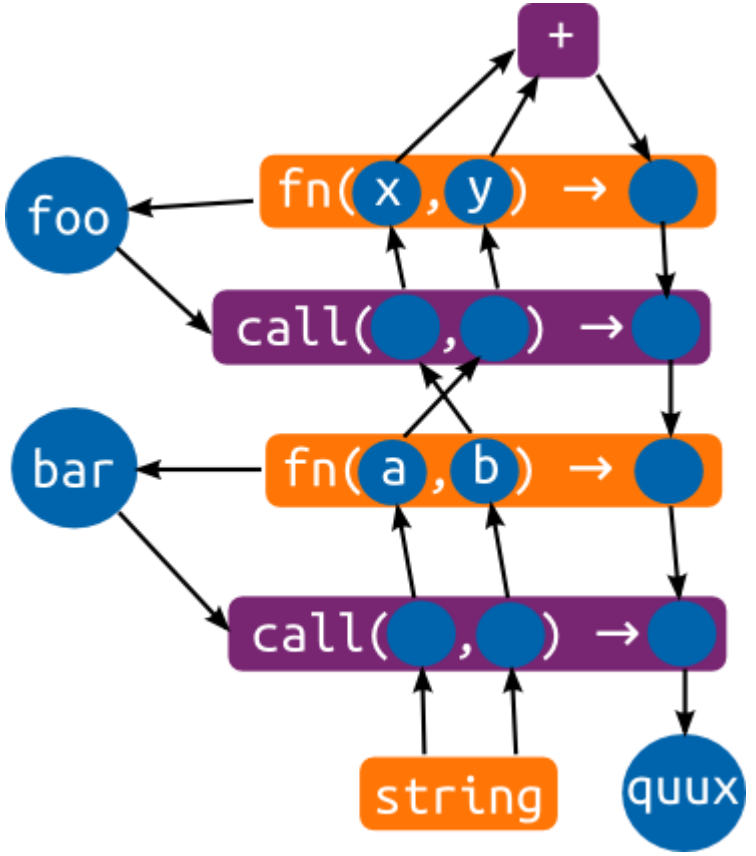


Type inference is a function which automatically specifies the type that can perform “inference” if the necessary minimum type is specified (Figure). If this function is used, the advantage where errors can be checked at an early stage can be realized, without specifying the type for all of the variables and functions. [**Java 8, Swift, Haskell, Scala**]

Type Propagation

```
function foo(x, y) { return (x + y); }
function bar(a, b) { return foo(b, a); }
var quux = bar("goodbye", "hello");
```

Type Propagation:
You can see the function types, as orange boxes, containing (references to) abstract values. Function declarations will cause such types to be created, and added to the variable that names the function. The purple boxes are propagation strategies. There are two calls in the program, corresponding to the two purple call boxes. At the top is a simple box that handles the + operator. If a string is propagated to it, it'll output a string type, and if two number types are received, it'll output a number type.



Advantages of Duck Typing

LECTURE 5

Polymorphic Method Design

Can we write a makelist function to replace

list(collection_data_type)

Key Point:

- Use built-in functions, language structures, and operators to achieve the polymorphism you want to achieve

list(collection_type)

Generic Method

```
>>> list([1, 2, 3])          # you can pass it a list
[1, 2, 3]
>>> list((1, 2, 3))          # you can also pass it a tuple
[1, 2, 3]
>>> list({'a', 'b', 'c'})    # or even a set
['b', 'a', 'c']              # (remember that sets are not ordered)
```

Polymorphic Language Structure (for-loop)

```
for x in [1, 2, 3]:
    print(x)

1
2
3
```

```
for x in (1, 2, 3):
    print(x)

1
2
3
```

```
# range() return tuple
for x in range(5):
    print(x)
1
2
3
4
5

for x in 3:
        print(x)

Traceback (most recent call last):
  File "", line 1, in
    for x in 3:
TypeError: 'int' object is not iterable
```

Implementation of makelist

```
def makelist(items):
    the_list = []

    for x in items:
        the_list.append(x)

    return the_list
```

And if we try this function out, we'll see it hits the nail right on the head:

```
>>> makelist([1, 2, 3])
[1, 2, 3]
>>> makelist((1, 2, 3))
[1, 2, 3]
>>> makelist({'a', 'b', 'c'})
['b', 'a', 'c']
>>> makelist(range(5))
[0, 1, 2, 3, 4]
```

Function Closure

LECTURE 6

Function Closure

A **function closure** is an object (interface) that have the same **polymorphic** method so that, the object can be passed as a parameter to a **generic method** (which operator on different object types with same polymorphic method).

The function closures can be run by the same generic function. (Same as the Java Calculus package developed by Dr. Eric Chou).

ducktyping2.py

```python
import math
class ZeroCalc:
    def calculate(self, n):
        return 0
class SquareCalc:
    def calculate(self, n):
        return n * n
class CubeCalc:
    def calculate(self, n):
        return n * n * n
class LengthCalc:
    def calculate(self, n):
        return len(n)
class SquareRootCalc:
    def calculate(self, n):
        return math.sqrt(n)
class MultiplyByCalc:
    def __init__(self, multiplier):
        self._multiplier = multiplier
    def calculate(self, n):
        return n * self._multiplier

def run_calcs(calcs: ['Calc'], starting_value):
    current_value = starting_value
    for calc in calcs:
        current_value = calc.calculate(current_value)
    return current_value


a1 = run_calcs([SquareCalc(), SquareCalc()], 4)
print(a1)
a2 = run_calcs([LengthCalc(), MultiplyByCalc(2)], 'Boo')
print(a2)
a3 = run_calcs([], 80)
print(a3)
a4 = run_calcs([MultiplyByCalc(3), LengthCalc(), SquareCalc()], 'Boo')
print(a4)
```

list of function closure.

A **function closure** is an **object** which has a polymorphic calculate method.

```
Run  ducktyping2
     C:\Python\Python36\python.exe
     256
     6
     80
     81
```