

Duck Typing

- The phrase "Duck Typing" originates from an old saying,
- If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.
- This means that - Even a non-duck entity that behaves like a duck can be considered a duck because emphasis is on behaviour.
- By analogy, for computing languages, the type of an object is not important so long as it behaves as expected.
- This behaviour is defined by the object's methods/properties/attributes while expectations are set by those who invoke the methods/properties/attributes.
- Duck Typing is a way of programming in which an object passed into a function or method supports all method signatures and attributes expected of that object at run time. The object's type itself is not important.
- Rather, the object should support all methods/attributes called on it. For this reason, duck typing is sometimes seen as "a way of thinking rather than a type system".
- In duck typing, we don't declare the argument types in function prototypes or methods. This implies that compilers can't do type checking.
- What really matters is if the object has the particular methods/attributes at run time. Duck typing is therefore often supported by dynamic languages.
- However, some static languages are beginning to "mimic" it via structural typing.

Advantage of Duck Typing

- It's been said that dynamic typing in general cuts down development time.
- There's less boilerplate code. Code is easier to hack.
- With static typing, compile-time checks slow down the development process.
- Static typing could be used at a later point to get better performance.
- Others recommend duck typing only for prototyping and never for production.
- While it's true that compile-time checks are useful to catch potential problems early, duck typing enforces following coding conventions, documentation and test-driven methodologies.
- Since duck typing isn't exactly a type system, it gives programmers flexibility.
- In Python, for example, common things are simpler to code.
- While static typed languages use interfaces, such interfaces may involve excessive refactoring of client code.

Consider below application which demonstrate concept of Duck Typing

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")
```

```
print("Demonstration of Duck Typing")
```

```
class Sparrow:
    def fly(self):
        print("Sparrow flying")
```

```
class Airplane:
    def fly(self):
```

```
print("Airplane flying")
```

```
class Whale:  
    def swim(self):  
        print("Whale swimming")
```

```
# The type of entity is not specified  
# We expect entity to have a callable named fly at run time  
def fun(entity):  
    entity.fly()
```

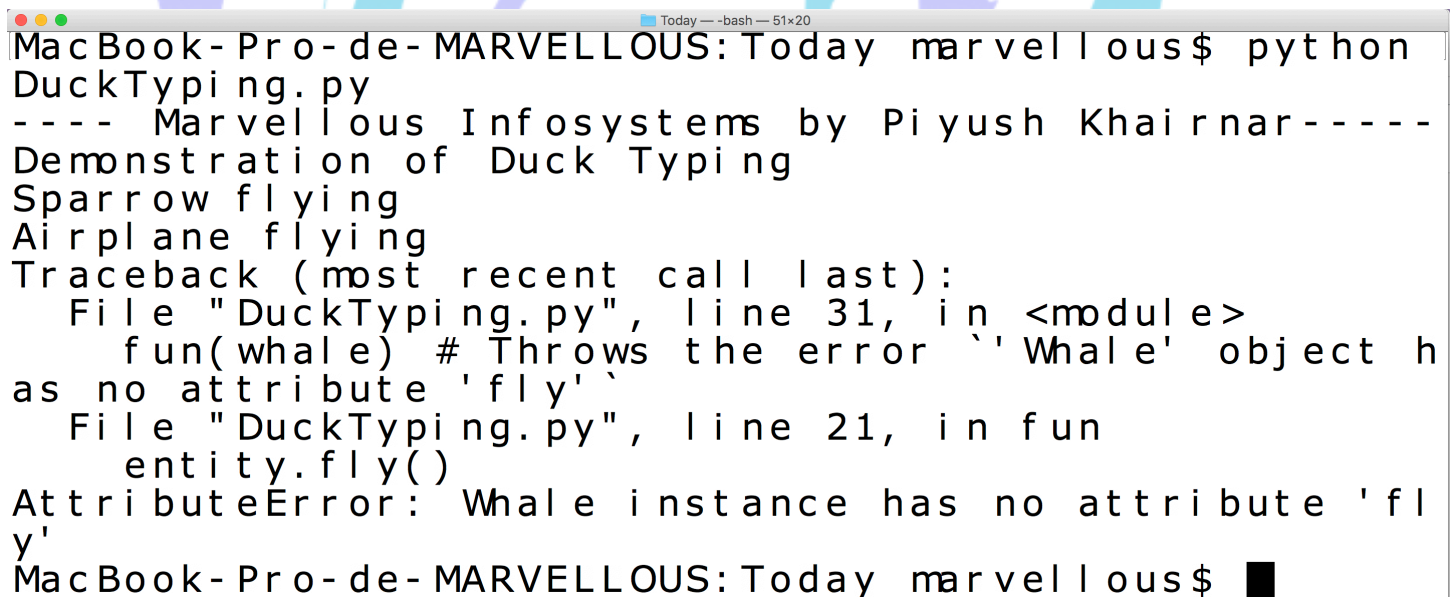
```
sparrow = Sparrow()  
airplane = Airplane()  
whale = Whale()
```

```
fun(sparrow) # prints `Sparrow flying`
```

```
fun(airplane) # prints `Airplane flying`
```

```
fun(whale) # Throws the error ` 'Whale' object has no attribute 'fly'`
```

Output of Above application



```
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python  
DuckTyping.py  
---- Marvellous Infosystems by Piyush Khairnar ----  
Demonstration of Duck Typing  
Sparrow flying  
Airplane flying  
Traceback (most recent call last):  
  File "DuckTyping.py", line 31, in <module>  
    fun(whale) # Throws the error ` 'Whale' object h  
as no attribute 'fly'`  
  File "DuckTyping.py", line 21, in fun  
    entity.fly()  
AttributeError: Whale instance has no attribute 'fl  
y'  
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```