Elaboration  
Haskell Concepts in Python

By  
Frederick Zech,  
Fabian Falco

Contents

[Immutable Data 3](#_Toc93256080)

[Type Variables 5](#_Toc93256081)

[Higher-Order Functions 6](#_Toc93256082)

[Lambda Expressions 8](#_Toc93256083)

[Currying 9](#_Toc93256084)

[Function Composition and Streaming 10](#_Toc93256085)

[Function Composition 10](#_Toc93256086)

[Streaming 11](#_Toc93256087)

[Algebraic Data Types 12](#_Toc93256088)

[Pure and Impure Side Effects 14](#_Toc93256089)

[State Monad in Python 15](#_Toc93256090)

# Immutable Data

The principle of Immutable Data compromises an object whose internal state can not be changed once it has been created. In Python there are built-in types that are immutable, nevertheless there are also mutable types like lists, sets, dictionaries and other user-defined classes.  
The immutable types of Python are Numbers, Strings, Tuples, Frozen Sets and immutable user-defined classes.

When creating a value the space in memory gets allocated and the variable points to that place in memory. If now the same variable gets used for another value the first value still exists in memory, but the binding to it is lost.

A picture containing text, electronics

Description automatically generated

Even though it seems that it is possible to change the value of a variable, Python will not allow it and will make a copy of the variable. This mechanism is called copy-on-modify.

x = 100  
print('Type: ', type(x), '\n',  
 'Memory id: ', id(x))  
#Type: <class 'int'>   
#Memory id: 140715477443440  
  
x = 200  
print('Type: ', type(x), '\n',  
 'Memory id: ', id(x))  
# Type: <class 'int'>  
# Memory id: 140715477446640  
  
x = float(x)  
print('Type: ', type(x), '\n',  
 'Memory id: ', id(x))  
# Type: <class 'float'>   
# Memory id: 1974826931504

In contrast to that, lists are mutable in python and are changeable without changing the position in memory.

myList = [1, 2, 3, 4, 5]  
print('Type: ', type(list), '\n',  
 'Memory id: ', id(list))  
# Type: <class 'type'>  
# Memory id: 140715477207088  
  
myList = [12, 33, 11, 42]  
print('Type: ', type(list), '\n',  
 'Memory id: ', id(list))  
# Type: <class 'type'>   
# Memory id: 140715477207088

<https://www.mygreatlearning.com/blog/understanding-mutable-and-immutable-in-python/>

<https://freecontent.manning.com/mutable-and-immutable-objects/>

# Type Variables

In Python variable type annotations are not enforced, so the variables do not need a declaration to reserve memory. By assigning a value to a variable the declaration happens automatically.

Furthermore, Python allows to change the type of a variable during runtime. It is possible for example to assign an integer value to a variable and change it to another value of type string.

At runtime a type can be checked by using the built in type() function. This can be very useful, as the declaration happens automatically, and the type can not be

# type variables  
value = 12  
print(type(value)) #<class 'int'>

value = "Haskell"  
print(type(value)) #<class 'str'>

value = ["H", "as", "kell"] #<class 'list'>  
print(type(value))

<https://docs.python.org/3.5/extending/newtypes.html>

<https://python-course.eu/python-tutorial/data-types-and-variables.php>

# Higher-Order Functions

A picture containing text, person

Description automatically generatedFunctions are called Higher Order Functions if they contain other functions as parameter or return a function as value. Python also supports the concept of Higher Order Functions.

There are different ways to define Higher Order Functions in Python:

* Passing functions as parameter for another function
* Returning a function from another function
* Using decorators as High Order Function

The most common way of Higher Order Functions is to pass them as parameter to another function. This can be accomplished by assigning a reference of a function to a variable. This variable can then be used to pass it as parameter to other functions.

# pass function to function  
def calculateSum(nums):  
 sum = 0  
 for num in nums:  
 sum += num  
 return sum  
  
  
def calculateProduct(nums):  
 prod = 1  
 for num in nums:  
 prod = prod \* num  
 return prod

def calculator(func, nums):  
 solution = func(nums)  
 print(solution)  
 return solution  
  
nums = [1, 2, 3, 4]  
calcSum = calculateSum  
calcProduct = calculateProduct  
calculator(calcSum, nums)  
calculator(calcProduct, nums)

<https://www.geeksforgeeks.org/higher-order-functions-in-python/>

<https://medium.com/analytics-vidhya/higher-order-functions-python-716f508a8f41>

<https://www.javatpoint.com/python-high-order-function>

# Lambda Expressions

Lambda Expressions are small anonymous functions without a name that take any number of arguments but can only have on expression. Lambda functions are mostly used as anonymous functions inside another function or as arguments to higher-order functions. Furthermore, they are also used when using built-in functions like filter() or map(). Lambda functions come in handy, when you want to save lines of code, as defining a helper function for only one use case can often be too much.

When using lambda in Python the following syntax must be followed:

lambda arguments: expression

def myfunc(n):  
 return lambda a: a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11)) #22  
print(mytripler(11)) #33  
  
# Program to filter out only the even items from a list  
my\_list = [1, 5, 4, 6, 8, 11, 3, 12]  
  
new\_list = list(filter(lambda x: (x % 2 == 0), my\_list))  
  
print(new\_list) #[4, 6, 8, 12]

<https://realpython.com/python-lambda/>

<https://www.programiz.com/python-programming/anonymous-function>

# Currying

With currying we can break down an evaluation of a function with multiple argument into evaluating a sequence of single-argument functions.

In Python this can be accomplished by chaining functions within other functions. The output of the inner function becomes the input of the outer function.

Additionally, it is possible to simplify currying with lambda expressions.

# Currying  
def add(a):  
 def add\_a(b):  
 return a + b  
 return add\_a  
  
# Currying with lambda  
add\_4 = lambda a: lambda b: lambda c: lambda d: a + b + c + d  
  
print(add(4)(5)) #9  
print(add\_4(2)(4)(3)(1)) #10

<https://python-course.eu/advanced-python/currying-in-python.php>

# Function Composition and Streaming

## Function Composition

Function composition combines two or more functions in such way that the output of a function becomes the input of the other one. For example, for the two functions F and G the composition is represented as F(G(x)). There, the output of G(x) becomes the input of F.

def pow(x):  
 return x \* x  
  
  
def double(x):  
 return x \* 2  
  
val = pow(double(5))  
  
print("Five times 2 is 10, and pow of 10 is 100.")  
print(val) # 100

<https://mathieularose.com/function-composition-in-python>

<https://www.geeksforgeeks.org/function-composition-in-python/>

## Streaming

With streams a sequence of data can be sent and received.

In python streams are included in the I/O library, where it is split into three main parts. I/O text, binary I/O, and raw I/O. In the example binary I/O is used.

# opens binary stream to file myfile.jpg  
f = open("myfile.jpg", "rb")  
  
# read bytes from the file  
b = f.read()  
  
# write bytes to the stream  
f.write(b"Some bytes to write")

<https://docs.python.org/3/library/io.html>

# Algebraic Data Types

Algebraic Data Types (ADTs) enables it to model structures in a comprehensive way which covers all possible outcomes. This makes the system less error-prone and helps on understanding the behaviour of a system.

ADTs are a way to declare concrete, recursive and abstract structures and make it possible to define the possibilities of these structures. (fixed set of cases) These structures are compositions of other types. At runtime it then will be checked of all the possibilities are considered.

Normally Python doesn’t support ADT as a way of defining types. Even though since Python 3 there is a possible way on using objects in the same style as ADTs.

This can be accomplished with the static type checking system of the mypy library, as well as the dataclasses library, which allows to define structures of data.

The idea behind this is to define each constructor as a dataclass and put the constructors together with a Union type. With isinstance tests it is then possible to do pattern matching on the result.

In the following there is an implementation of an algebraic data type in Haskell that can be used to show if a result is ok or ends up in a failure. The same got implemented in Python with the above-described methods.

In Haskell:

data Result = OK Int

              | Failure String

showResult :: Result -> String

showResult (OK result) = show result

showResult (Failure msg) = "Failure: " ++ msg

In Python:

@dataclass(frozen=True)  
class OK:  
 result: int  
  
@dataclass(frozen=True)  
class Failure:  
 msg: str  
  
Result = Union[OK, Failure]  
  
def assert\_never(x: NoReturn) -> NoReturn:  
 raise AssertionError("Unhandled type: {}".format(type(x).\_\_name\_\_))  
  
def showResult(r: Result) -> str:  
 if isinstance(r, OK):  
 return str(r.result)  
 elif isinstance(r, Failure):  
 return "Failure: " + r.msg  
 else:  
 assert\_never(r)  
  
ok = OK(123)  
fail = Failure("Failure")  
print(showResult(ok)) #123  
print(showResult(fail)) #Failure: Failure

<https://www.gidware.com/python-adts/>

<http://blog.ezyang.com/2020/10/idiomatic-algebraic-data-types-in-python-with-dataclasses-and-union/>

# Pure and Impure Side Effects

Pure functions do not use any external libraries, so the output for the same given input is always the same. As they are conceptually simple, they are also much easier to test. When writing pure functions in Python global statements have to be avoided.

Impure functions are all functions that communicate with the outside world and therefore have side effects.

In this example the pure function has no side effect because it changes the state of the program from 0 to 10. The impure function has side effects because it takes input from the user and prints it into the console.

\_state = 0  
  
  
def pure(x):  
 global \_state  
 \_state = 10  
  
  
def impure():  
 username = input("Enter username:")  
 print("Username is: " + username)  
  
  
print(\_state)  
pure(10)  
impure()  
print(\_state)

<https://stackoverflow.com/questions/20027087/how-to-judge-or-how-to-write-a-python-function-with-no-side-effects>

# State Monad in Python

1. Implement the State Monad in the selected language and demonstrate its use through a simple example such as tree labeling.

The State monad contains an immutable state that is passed through a chain of computations, and the output gets a new state appended to it. It enables us to manage the state of a program effectively in a functional way.

In the following our implementation of a Python State Monad is demonstrated.  
With the put method we can update the state to a new state. Get gives us a state object with the current state. With the bind function we can concatenate different computations that will run after each other in a sequence.

from typing import Callable, Tuple, Any, TypeVar, Generic  
  
from basic import Unit  
from functor import Functor  
from monad import Monad  
  
TState = TypeVar("TState")  
TSource = TypeVar("TSource")  
TResult = TypeVar("TResult")  
  
  
class State(Generic[TSource, TState]):  
 *"""The state monad.  
 Wraps stateful computations. A stateful computation is a function  
 that takes a state and returns a result and new state:  
 state -> (result, state')  
 """* def \_\_init\_\_(self, fn: Callable[[TState], Tuple[TSource, TState]]) -> None:  
 *"""Initialize a new state.  
 Keyword arguments:  
 fn -- State processor.  
 """* self.\_fn = fn  
  
 @classmethod  
 def unit(cls, value: TSource) -> "State[TSource, TState]":  
 *r"""Create new State.  
 The unit function creates a new State object wrapping a stateful  
 computation.  
 State $ \s -> (x, s)  
 """* return cls(lambda state: (value, state))  
  
 def map(self, mapper: Callable[[TSource], TResult]) -> "State[TResult, TState]":  
 def \_(a: Any, state: Any) -> Tuple[Any, Any]:  
 return mapper(a), state  
  
 return State(lambda state: \_(\*self.run(state)))  
  
 def bind(self, fn: Callable[[TSource], "State[TState, TResult]"]) -> "State[TResult, TState]":  
 *r"""m >>= k = State $ \s -> let (a, s') = runState m s  
 in runState (k a) s'  
 """* def \_(result: Any, state: Any) -> Tuple[Any, Any]:  
 return fn(result).run(state)  
  
 return State(lambda state: \_(\*self.run(state)))  
  
 @classmethod  
 def get(cls) -> "State[TState, TState]":  
 *r"""get = state $ \s -> (s, s)"""* return State(lambda state: (state, state))  
  
 @classmethod  
 def put(cls, new\_state: TState) -> "State[Tuple, TState]":  
 *r"""put newState = state $ \s -> ((), newState)"""* return State(lambda state: (Unit, new\_state))  
  
 def run(self, state: TState) -> Tuple[TSource, TState]:  
 *"""Return wrapped state computation.  
 This is the inverse of unit and returns the wrapped function.  
 """* return self.\_fn(state)  
  
 def \_\_call\_\_(self, state: Any) -> Tuple:  
 return self.run(state)  
  
  
assert issubclass(State, Functor)  
assert issubclass(State, Monad)

from stateMonad import State  
  
unit = State.unit  
put = State.put  
get = State.get  
  
  
class Tree:  
 def \_\_init\_\_(self, leftTree, value, rightTree):  
 self.leftTree = None  
 self.rightTree = None  
 self.value = value  
  
 def add(self, val):  
 if self.value is None:  
 self.value = val  
 else:  
 self.\_add(val, self.value)  
  
 def \_add(self, val, node):  
 leftTree = self.leftTree  
 rightTree = self.rightTree  
 nodeValue = node  
 if isinstance(node, Tree):  
 nodeValue = node.value  
 leftTree = node.leftTree  
 rightTree = node.rightTree  
 if val < nodeValue:  
 if leftTree is not None:  
 self.\_add(val, leftTree)  
 else:  
 if isinstance(node, Tree):  
 node.leftTree = Tree(None, val, None)  
 else:  
 self.leftTree = Tree(None, val, None)  
 else:  
 if rightTree is not None:  
 self.\_add(val, rightTree)  
 else:  
 if isinstance(node, Tree):  
 node.rightTree = Tree(None, val, None)  
 else:  
 self.rightTree = Tree(None, val, None)  
  
 def printTree(self):  
 if self.value is not None:  
 self.\_printTree(self)  
  
 def \_printTree(self, node):  
 if node is not None:  
 self.\_printTree(node.leftTree)  
 print(str(node.value) + ' ')  
 self.\_printTree(node.rightTree)  
  
 def find(self, val):  
 if self.value is not None:  
 return self.\_find(val, self.value)  
 else:  
 return None

def \_find(self, val, node):  
 if val == self.value:  
 return node  
 elif val < self.value and self.leftTree is not None:  
 return self.\_find(val, self.leftTree)  
 elif val > self.value and self.rightTree is not None:  
 return self.\_find(val, self.rightTree)  
  
 def getRoot(self):  
 return self  
  
 def deleteTree(self):  
 # garbage collector will do this for us.  
 self.value = None  
 self.leftTree = None  
 self.rightTree = None  
  
 def isLeaf(self):  
 if self.leftTree is None and self.rightTree is None:  
 return True  
 else:  
 return False  
  
  
tree = Tree(None, "c", None)  
tree.add("b")  
tree.add("e")  
tree.add("d")  
tree.add("f")  
tree.add("a")  
tree.add("g")  
  
def labelTreeMonad(fstNode) -> State:  
 if fstNode is not None and not fstNode.isLeaf():  
 state = get().bind(  
 lambda value: labelTreeMonad(fstNode.leftTree).bind(lambda \_: labelTreeMonad(fstNode.rightTree)).bind(  
 lambda \_: put(value + 1).bind(lambda \_: unit(fstNode.value))))  
  
 print(state)  
 return state  
 elif fstNode is not None and fstNode.isLeaf():  
 newState = get().bind(lambda value:  
 put(value + 1).bind(  
 lambda \_: unit(fstNode.value)))  
 print(newState)  
 return newState  
 elif fstNode is None:  
 newState = get()  
 return newState  
  
  
result = labelTreeMonad(tree).run(0)

<https://gaius.tech/2010/09/06/on-monads/>

<https://medium.com/swlh/monads-in-python-e3c9592285d6>