Functional Programming inside OOP?

It’s possible with Python
Carlos Villavicencio

- Ecuadorian 🇪🇨
- Currently: Python & TypeScript
- Community leader
- Martial arts:剣道、居合道
- Nature photography enthusiast

Cayambe Volcano, 2021.
why_functional_programming

- Easier and efficient
- Divide and conquer
- Ease debugging
- Makes code simpler and readable
- Also easier to test
Functions were first-class objects from design.
Users wanted more functional solutions.
1994: map, filter, reduce and lambdas were included.
In Python 2.2, lambdas have access to the outer scope.

“Not having the choice streamlines the thought process.”
- Guido van Rossum.

The fate of reduce() in Python 3000

>>> has_django_fp()

def is_valid(self):
    """
    Returns True if form.errors is empty for every form in self.forms.
    """
    if not self.is_bound:
        return False
    # We loop over every form.errors here rather than short circuiting on the 
    # first failure to make sure validation gets triggered for every form.
    forms_valid = True
    for errors in self.errors:
        if bool(errors):
            forms_valid = False
    return forms_valid and not bool(self.non_form_errors())

def is_valid(self):
    """
    Return True if every form in self.forms is valid.
    """
    if not self.is_bound:
        return False
    # Accessing errors triggers a full clean the first time only.
    self.errors
    # List comprehension ensures is_valid() is called for all forms.
    # Forms due to be deleted shouldn't cause the formset to be invalid.
    forms_valid = all(
        for form in self.forms
            if form.is_valid() and
            if not (form.can_delete and form._should_delete_form(form))
    )
    return forms_valid and not self.non_form_errors()
Immutability

An immutable object is an object whose state cannot be modified after it is created.

Booleans, strings, and integers are immutable objects.

List and dictionaries are mutable objects.

Thread safety
```python
def update_list(value: list) -> None:
    value += [10]

>>> foo = [1, 2, 3]
>>> id(foo)
4479599424
>>> update_list(foo)
>>> foo
[1, 2, 3, 10]
>>> id(foo)
4479599424

def update_number(value: int) -> None:
    value += 10

>>> foo = 10
>>> update_number(foo)
>>> foo
10 🤔
```
>>> immutability

def update_number(value: int) -> None:
    print(value, id(value))
    value += 10
    print(value, id(value))

>>> foo = 10
>>> update_number(foo)
10 4478220880
20 4478221200
>>> foo
10

Decorators

They are functions which modify the functionality of other functions.

Higher order functions.

Closures?
>>> decorators

def increment(x: int) -> int:
    return x + 1

>>> increment(2)
3
>>> decorators

def increment(x: int) -> int:
    return x + 1

def double_increment(func: Callable) -> Callable:
    def wrapper(x: int):
        r = func(x)  # func is saved in __closure__
        y = r * 2
        return y
    return wrapper
>>> decorators

@double_increment
def increment(x: int) -> int:
    return x + 1

>>> increment(2)
6
>>> increment.__closure__[0].cell_contents
<function increment at 0x7eff362cf940>
>>> increment.__closure__[0].cell_contents(2)
3
Partial application of functions

They reduce the number of arguments that any function takes.

Makes functions easier to compose with others.
>>> partial_application

def get_url(url: str, role: str) -> str:
    pass

from functools import partial

going_admin_url = partial(get_url, "admin")
>>> partial_application

```python
import re
from functools import partial

e-mail_match = partial(re.match, r'^/\.|\_\-\+[@]/\.|\_\-\+\[.]\w\{2,3\}$')

url_match = partial(re.match, r"(?i)\b((?:https?://|www\d\d?\[.\]1\d\d\d\[.\]a-z\d-9\[.\]a-z\{2,4\}/)?(\^[\s()<>\]\{\}[\s()<>\]\{\}\[\s`!(){};:'".,<>?«»”‘’’']+)\}\{\}+\{\}+\{\}\}+\{\}(\^[\s()<>\]\{\}[\s()<>\]\{\}\[\s`!(){};:'".,<>?«»”‘’’’\])")
```
Lazy Evaluation

It holds the evaluation of an expression until the value is finally needed.

Reduce the memory footprint.
```python
def generator():
    i = 1
    while True:
        yield i
        i += 1
```
>>> lazy_evaluation

    with open(filename, 'r') as f:
        for line in f:
            process(line)
Type Annotations

PEP 484
Available since Python 3.5
Reduce bugs at runtime
Improves readability
Structural Pattern Matching

PEP-634
Available since Python 3.10
It doesn’t work as C or JavaScript
It’s a declarative approach!
```python
# point is an (x, y) tuple[int, int]
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y) if x == y:  # guard
        print(f"X=Y={x}"),
    case (x, y):
        print(f"X={x}, Y={y}"),
    case _:  # wildcard
        raise ValueError("Not a point")
```
>>> structural_pattern_matching

# test_variable is a tuple[str, Any, int]
match test_variable:
    case ('warning', code, 40):
        print("A warning has been received."")
    case ('error', code, _):
        print(f"An error {code} occurred.")
Other Functional Programming Patterns

When Python doesn’t offer a way to do it, you can always implement it.

Currying

Composition
If a function $f_n$ takes $n$ arguments, then you can turn that into a function $c_n$ which takes one argument and returns a function $c_{n-1}$ that takes $n-1$ arguments, and has access to the argument that was passed to $c_n$ (hence $c_{n-1}$ is a closure)

https://sagnibak.github.io/blog/python-is-haskell-currying/
>>>currying

def f_5(a: int, b: int, c: int, d: int, e: int) -> int:
    return a + b + c + d + e
>>> currying

def c_5(a: int) -> Callable:
    def c_4(b: int) -> Callable:
        def c_3(c: int) -> Callable:
            def c_2(d: int) -> Callable:
                def c_1(e: int): int:
                    return f_5(a, b, c, d, e)
                return c_1
            return c_2
        return c_3
    return c_4

Then, f_5(1, 2, 3, 4, 5) == c_5(1)(2)(3)(4)(5)
>>> currying

@curry(num_args=5)
def c_5(a: int, b: int, c: int, d: int, e: int) -> int:
    a + b + c + d + e

https://sagnibak.github.io/blog/python-is-haskell-currying/
```bash
>>> cat .env | grep DEBUG
ASSETS_DEBUG=True
SENTRY_DEBUG=False
```
sortByDateDescending = reverse . sortByDate
```python
def compose2(f, g):
    return lambda x: f(g(x))
```
```python
import functools

def compose(*functions):
    def compose2(f, g):
        return lambda x: f(g(x))
    return functools.reduce(compose2, functions, lambda x: x)

>>> composition
```
>>> composition

def td(val: str) -> str:
    return f"<td>{val}</td>"

def tr(val: str) -> str:
    return f"<tr>{val}</tr>"

def table(val: str) -> str:
    return f"<table>{val}</table>"

>>> one_cell_table = compose(table, tr, td)
>>> one_cell_table("something")
'<table><tr><td>something</td></tr></table>"
Everything we covered before makes our tests easier.
>>>import unittest

"Code that is hard to test is not good code"
- Joe Eames.

https://dev.to/leolanese/making-unit-test-fun-again-with-functional-programming-4g8m
>>> import unittest

“The outcome of a function is dependent only on the input and nothing else”
- Unknown author.

https://dev.to/leolanese/making-unit-test-fun-again-with-functional-programming-4g8m
>>> import unittest

“OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.”
- Michael Feathers.

https://dev.to/leolanese/making-unit-test-fun-again-with-functional-programming-4g8m
Thank you for your attention 😊