FUNCTIONAL PROGRAMMING IN PYTHON

Niels Hendrikx – Team Nevo

PROGRAM

- Introduction
- What is FP
- FP in Python
- Concepts
  - Pure functions
  - Immutabillity
  - First-class functions
  - Coperhensions
  - Higher-order functions
  - Anonymous functions
  - Lazy evaluation
    - Iterators / Generators

INTRODUCTION

- Software engineer @ Team NeVo
  - Python
- 31 Years, Amsterdam
- FP @ UU / CS in Haskell
  - Enjoyed
  - Not that applicable in real world applications
  - Apply certain concepts

INTRODUCTION - HISTORY

- FP comes from mathematical logic / lambda calculus
- Alonzo Chruch – 1930 – Lambda calculus
  - *A system for expressing computation using variable binding and subsitution*
- *John McCarthy – 1958 – Lisp*
- The first programming language to adopt this is Lisp, also:
  - Recursion
  - If-else

INTRODUCTION - HISTORY

Number	definition Function	expression Lambda
0	$0\ f\ x = x$	$0 = \lambda f.\lambda x.x$
1	$1\ f\ x = f\ x$	$1 = \lambda f.\lambda x.f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f.\lambda x.f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f.\lambda x.f\ (f\ (f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\ f\ x = f^n\ x$	$n = \lambda f.\lambda x.f^n\ x$

INTRODUCTION - HISTORY

The following function reverses a list. (Lisp's built-in *reverse* function does the same thing.)

```
(defun -reverse (list)
    (let ((return-value))
      (dolist (e list) (push e return-value))
      return-value))
```

WHAT IS FP?

In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can. This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

DECLARATIVE - IMPERATIVE

- OOP Imperative
  - Step-by-step instruction to achieve result
  - Declare variables
  - Alter state with methods, loops, etc.
- FP Declarative
  - Describes desired outcome
  - Functions are pure
    - No side effect: It does not change anything outside its scope
    - Determenistic
- Examples: Create a list of even numbers from 1-10

DECLARATIVE - IMPERATIVE

```
def get_even_numbers_imperative():
    even_numbers = []
    for i in range(11):
        if i % 2 == 0:
            even_numbers.append(i)
    return even_numbers
```

DECLARATIVE - IMPERATIVE

```python
def get_even_numbers_declarative(start=1, end=10):
    if start > end:
        return []

    if start % 2 == 0:
        return [start] + get_even_numbers_declarative(start + 1, end)

    return get_even_numbers_declarative(start + 1, end)
```

DECLARATIVE - IMPERATIVE

```
list(filter(lambda num: num % 2 == 0, range(start, end + 1)))
```

FP IN PYTHON?

- Can we FP in Python?
- Should you?

FIRST CLASS FUNCTIONS

- To be able to program functional, functions must be treated as first class citizens.
  1. Take another function as an argument *(callback)*
  2. Return another function to its caller
- Suprize, In Python everything is in object, including functions.

FIRST CLASS FUNCTIONS

Python

```python
>>> def func():
...     print("I am function func()!")
...

>>> func()
I am function func()!

>>> another_name = func
>>> another_name()
I am function func()!
```

FIRST CLASS FUNCTIONS

Python

```
>>> def func():
...     print("I am function func()!")
...

>>> print("cat", func, 42)
cat <function func at 0x7f81b4d29bf8> 42

>>> objects = ["cat", func, 42]
>>> objects[1]
<function func at 0x7f81b4d29bf8>
>>> objects[1]()
I am function func()!

>>> d = {"cat": 1, func: 2, 42: 3}
>>> d[func]
2
```

FIRST CLASS FUNCTIONS

Python

```
>>> def inner():
...     print("I am function inner()!")
...

>>> def outer(function):
...     function()
...

>>> outer(inner)
I am function inner()!
```

FP IN PYTHON?

- Can we FP in Python? YES!
- Should you?
  - Advantages
  - Disadvantages

ADVANTAGES

- It seems we could functional progam in Python, but why do we want this?
- The behavior of pure functions are desribed by only inputs/outputs without side effects
  - Easier debugging
  - Easier testing
  - Better readabillity/understandabillity
  - Provabillity your program is 100% correct
- Routines that don’t cause side-effects and don’t have mutual shared memory (state) are easily run in parallel
- Lazy evaluation
  - Efficient when doing calculations on large datasets
- Python may not be a functional language, hower the creators adopted many functional programming concepts.
- Fun & elegant

DISADVANTAGES

- Not actually functional
  - Immutabillity is not enforced in Python
  - State is everywhere
    - Built-in functions, libaries and applications in Python will have state
  - Type safety not enforced
- Readabillity can be difficult for people without FP experience
- Conurrency in python
  - OvErHeAd
- But the concepts are language agnostic

PURE FUNCTIONS

- Avoid impure functions, but what is a pure function?
- A **pure function**
  - Same input/output
  - Think of it like a mathematical function.

  No Side Effects It doesn't modify a global variable, change the state of an object, perform I/O operations (like printing to the console or writing to a file), or make a network request.
- **Why?**
  - **Predictable**: Because of their nature, pure functions are easy to reason about and test, as their behavior is entirely dependent on their arguments.
  - **Concurrency / memory safe**
  - **Memoization,** Determenistic by nature → efficient caching
- **You need impurity?** Techniques like separation of pure and impure, monads, abstractions etc.

PURE FUNCTIONS

```python
name: str = "Alice"

def greet(person_name: str) -> str:
  return f"Hello, {person_name}!"
```

PURE FUNCTIONS

```python
from typing import Dict, List, Any

database: Dict[str, List[Any]] = {
  "users": []
}


def add_user(user: str) -> None:
  database["users"].append(user)
```

PURE FUNCTIONS

```python
import datetime

def get_timestamp() -> datetime.datetime:
  return datetime.datetime.now()
```

PURE FUNCTIONS

```python
from typing import NamedTuple

class BankAccountPure(NamedTuple):
    owner: str
    balance: float

def create_new_account_with_deposit(
    account: BankAccountPure, amount: float
) -> BankAccountPure:
    """
    Creates and returns a new account object with the updated balance.
    """
    if amount <= 0:
        return account
    return BankAccountPure(
        owner=account.owner,
        balance=account.balance + amount
    )
```

PURE FUNCTIONS

```
from typing import Dict, Any

TAX_RATE: float = 0.08

def apply_discount(product: Dict[str, Any], discount_rate: float) -> float:
    """
    Calculates the final price of a product after a discount and a global tax rate.
    """
    if 'price' not in product:
        raise ValueError("Product must have a price key.")

    final_price = product['price'] * (1 - discount_rate)
    return final_price * (1 + TAX_RATE)
```

PURE FUNCTIONS

```python
from typing import Dict, Any

def calculate_final_price(
    product: Dict[str, Any],
    discount_rate: float,
    tax_rate: float
) -> float:
    if 'price' not in product:
        raise ValueError("Product must have a price key.")

    final_price = product['price'] * (1 - discount_rate)
    return final_price * (1 + tax_rate)
```

IMMUTABILLTY

- **Immutability** is the principle that once an object is created, it cannot be changed. Instead of modifying an object in place, you create a new one with the desired changes.
- **Why it's important**: Mutability can lead to unexpected behavior and bugs, especially in concurrent programming, where multiple threads might try to modify the same data simultaneously.
- **Python's built-in immutable types**: **tuples**, **strings**, and **integers** are all immutable. **lists** and **dictionaries** are mutable.

IMMUTABILLTY

```python
# Immutable
my_tuple = (1, 2, 3)
# my_tuple[0] = 5  # This will raise a TypeError

# Mutable
my_list = [1, 2, 3]
my_list[0] = 5  # This works, my_list is now [5, 2, 3]
```

```
>>> word_counter = letter_counter = 0
>>> id(word_counter)
4343439560
>>> id(letter_counter)
4343439560

>>> word_counter += 1
>>> word_counter
1
>>> id(word_counter)
4343439592

>>> letter_counter
0
>>> id(letter_counter)
4343439560
```

FIRST-CLASS FUNCTIONS

- In Python, functions are **first-class citizens**, which means they can be treated just like any other variable.
- **Assign to variables**: You can assign a function to a variable.
- **Pass as arguments**: You can pass a function as an argument to another function.
- **Return from functions**: You can return a function as the result of another function.

FIRST-CLASS FUNCTIONS

```python
def greet():
    return "Hello, World!"

# Assign a function to a variable
say_hello = greet

# Pass a function as an argument
def call_func(func):
    return func()

result = call_func(say_hello) # Returns "Hello, World!"
```

COPERHENSIONS

- **Comprehensions** provide a concise and readable way to create new sequences (like lists, dictionaries, and sets) from existing ones.
- **List Comprehensions**: The most common type. Creates a new list.
- **Dictionary Comprehensions**: Creates a new dictionary.
- **Set Comprehensions**: Creates a new set.
- **Not strictly functional**, but declarative
- **Less overhead** compared to a loop

```
# List comprehension
squares = [x**2 for x in range(10)]  # Creates [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Dictionary comprehension
squares_dict = {x: x**2 for x in range(5)}  # Creates {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

ANONYMOUS FUNCTIONS

- **Anonymous functions**, also known as **lambda functions**, are small, single-expression functions that don't need a name. They are often used as a concise way to create a function object on the fly, especially when passing a function as an argument to a higher-order function.
- **Syntax**: lambda arguments: expression
- **Limitations**: They can only contain one expression.

ANONYMOUS FUNCTIONS

```
# A lambda function to add two numbers
add = lambda x, y: x + y
print(add(5, 3))  # Prints 8

# Using lambda with map()
numbers = [1, 2, 3]
doubled = list(map(lambda x: x * 2, numbers))  # [2, 4, 6]
```

HIGHER-ORDER FUNCTIONS

- A **higher-order function** is a function that either takes one or more functions as arguments, returns a function, or both.
- **map()**: Applies function to every element in iterable.
- **filter()**: Applies predicate to every element in iterable.
- The **reduce():** Takes a function that describes how to aggregate a given iterable.

HIGHER-ORDER FUNCTIONS - MAP

- Programming without an loop
- **Map(func: Callable[[A], B], it: Iterable[A]) → Iterable[B]**
- **func:** transformation function

- Avoiding side effects & immutability issues
- Map can be faster as a loop due built in optimization
- Map has less memory footprint → lazy evaluation
- Works well with built in multithreading/concurrency libraries

HIGHER-ORDER FUNCTIONS - MAP

```
>>> string_it = ["processing", "strings", "with", "map"]
>>> list(map(str.capitalize, string_it))
['Processing', 'Strings', 'With', 'Map']

>>> list(map(str.upper, string_it))
['PROCESSING', 'STRINGS', 'WITH', 'MAP']

>>> list(map(str.lower, string_it))
['processing', 'strings', 'with', 'map']
```

HIGHER-ORDER FUNCTIONS - FILTER

- Filtering data
- **filter(func: Callable[[A], bool], it: Iterable[A]) → Iterable[A]**
- **Func:** fun(element: a) → bool

- Avoiding side effects & immutability issues
- Applying a predicate to each object in an iterable
- filter can be faster
- filter has less memory footprint

HIGHER-ORDER FUNCTIONS - FILTER

```
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> def is_even(number):
...     return number % 2 == 0
...

>>> even_numbers = list(filter(is_even, numbers))
>>> even_numbers
[10, 6, 50]
```

HIGHER-ORDER FUNCTIONS - REDUCE

- Called **fold** in other functional languages
- Can be imported from itertools
- **Reduce**(Func: Callable[[A, B], A], Iter: Iterable[B], Initializer: A) → A
  - Callable:[[A, B], A]: A function that takes 2 arguments, and returns 1
  - **Iter: Iterable[B]**:
  - **Initializer: A** (Optional): This represents an initial value for the accumulator, and its type is A. This is the same type as the accumulator and the final return value.

HIGHER-ORDER FUNCTIONS - REDUCE

```
>>> from functools import reduce

>>> numbers = [1, 2, 3, 4]

>>> reduce(lambda a, b: a + b, numbers)
10
```

HIGHER-ORDER FUNCTIONS - REDUCE

```
reduce(function: Callable[[_T, _T], _T], iterable: Iterable[_T]) -> _T
```

Python

```python
from functools import reduce
reduce(function, iterable[, initializer])
```

- `function`: A function that takes exactly two arguments.
- `iterable`: The sequence of items to be reduced.
- `initializer`: An optional value that is placed before the items of the iterable in the calculation.

HIGHER-ORDER FUNCTIONS - REDUCE

```
reduce(function: Callable[[_T, _T], _T], iterable: Iterable[_T]) -> _T
```

```
>>> from functools import reduce

>>> numbers = [0, 1, 2, 3, 4]

>>> reduce(my_add, numbers, 100)
100 + 0 = 100
100 + 1 = 101
101 + 2 = 103
103 + 3 = 106
106 + 4 = 110
110
```

LAZY EVALUATION

- **Lazy evaluation** is a strategy that delays the computation of an expression until its value is needed.
- **What it means**: Instead of calculating and storing a large sequence in memory all at once, you generate the values one at a time as you iterate over them.
- **Benefits**: It's memory-efficient and can improve performance, especially when dealing with large datasets or infinite sequences.
- **Iterators** and **generators** are key to implementing lazy evaluation in Python.

LAZY EVALUATION - ITERATORS

- **Iterators**: An object that can be iterated upon. It has a __next__() method that returns the next item in the sequence and a __iter__() method that returns the iterator itself.

```
# A lazy object (a map iterator)
lazy_squared = map(lambda x: x * x, range(1000000))

# The actual computation only happens when we iterate
first_five = [next(lazy_squared) for _ in range(5)]
```

LAZY EVALUATION - GENERATOR

- **Generators**: A simple and powerful tool for creating iterators. They are defined just like a normal function but use the yield keyword instead of return. When a generator function is called, it returns a generator object without running the code inside. The code runs only when next() is called on the object.

```python
# A generator function
def countdown(n):
    print("Starting countdown...")
    while n > 0:
        yield n
        n -= 1

# Creating a generator object
counter = countdown(3)

# Iterating and printing values lazily
print(next(counter))  # Prints "Starting countdown..." then 3
print(next(counter))  # Prints 2
print(next(counter))  # Prints 1
```

SOME PRACTICE

- https://github.com/nielsyh/functional-programming-python
- Some practice exercises, resolve without loops

1. Imperative to functional
2. Comprehensions
3. Pattern_matching
4. Higher order functions (No loops or comprehensions)
5. Combination (No loops or comprehensions)

Try it without AI ;)

I will upload answers later..