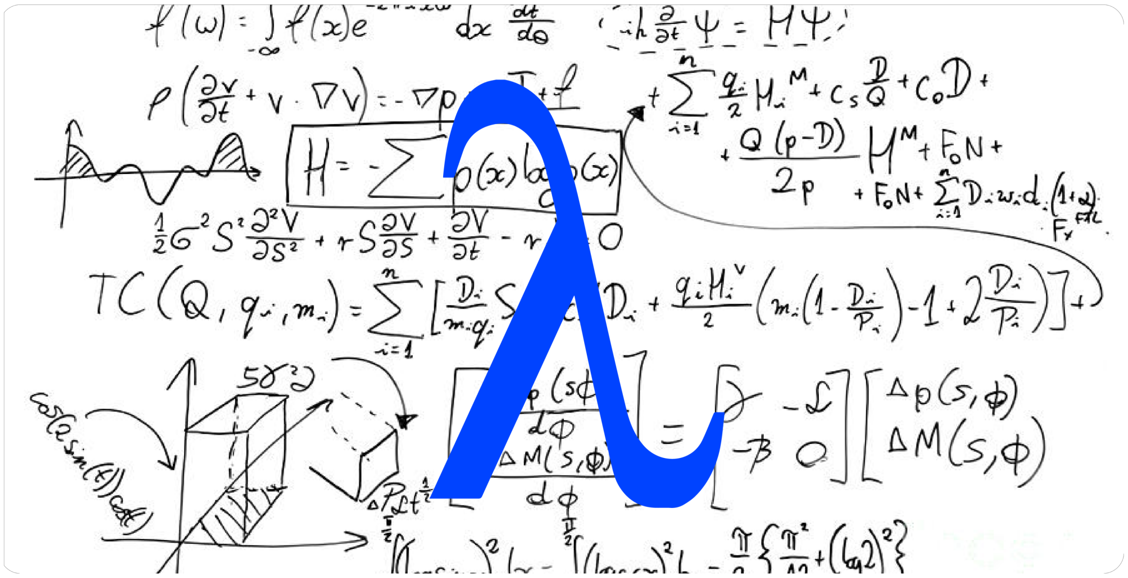
functional-python

April 10, 2022

0.1 Functional Programming in Python

By Vic Kumar | https://github.com/vickumar1981/functional_python



Salt Lake City, UT | PyCon 2022

0.2 About Me

Software Developer at Excella, Inc.

https://excella.com

Technology Consulting firm based in Arlington, VA

Modern Software Delivery, AI and Data Analytics, Agile Transformation

0.3 Projects

0.3.1 Goto Code | https://gotocode.io

Remote interviews, coding problems, hackathons, and code katas

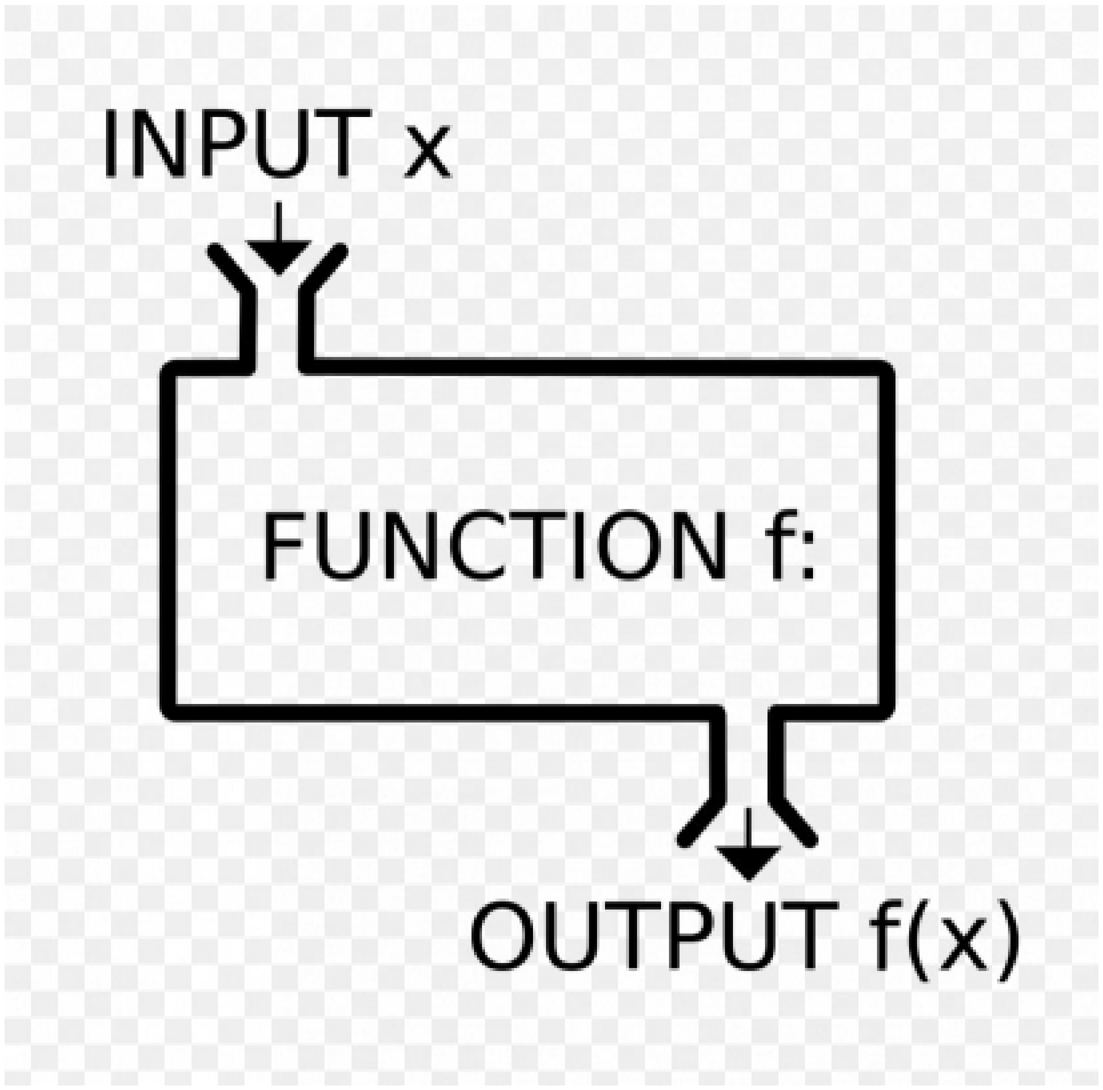
0.3.2 HBCU Digital | https://hbcudigital.com

HBCU DIGITAL

Stay current with all the latest news, photos, videos, scores and more on Historically Black College and Universities. Stream exclusive live sports and Originals that celebrate Black voices.

0.4 What is functional programming?

Functional programming (FP) is based on a simple premise with far-reaching implications: we construct our programs using only pure functions—in other words, functions that have no side effects.



What are side effects? A function has a side effect if it does something other than simply return a result, for example:

- Modifying a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

0.5 What is referential transparency?

An expression e is referentially transparent if, for all programs p, all occurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p. A function f is pure if the expression f(x) is referentially transparent for all referentially transparent x.

[1]:
```
from typing import List

def add_numbers(numbers: List[int]) -> int:
    sum = 0
    for n in numbers:
        sum = sum + n
    return sum

add_numbers([1, 5, 6, 8])
```

[1]:
```
20
```

[2]:
```
from typing import List

def add_numbers(numbers: List[int]) -> int:
    # if sum = 0, we should be able to replace sum on the RHS with 0 and get␣
 ↪the same result
    sum = 0
    for n in numbers:
        sum = 0 + n
    return sum

add_numbers([1, 5, 6, 8])
```

[2]: 8

[3]:
```python
# Let's use recursion

def add_numbers2(numbers: List[int]) -> int:
    return numbers[0] if len(numbers) == 1 else numbers[0] +␣
 ↪add_numbers2(numbers[1:])

add_numbers2([1, 5, 6, 8])
```

[3]: 20

[4]:
```python
# Let's use a higher-order function

from functools import reduce

def add_numbers3(numbers: List[int]) -> int:
    return reduce(lambda a, b: a + b, numbers)

add_numbers3([1, 5, 6, 8])
```

[4]: 20

0.5.1 pyEffects: Let's add some classes to deal with common behaviors

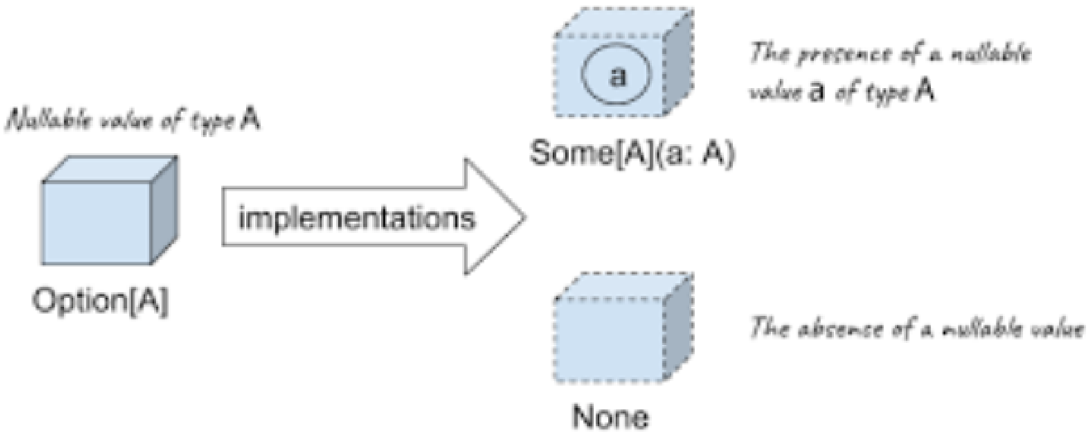
Classes: Option, Try, Either, and Future https://github.com/vickumar1981/pyeffects/

Read the Docs

[5]:
```
!pip install pyeffects
```

```
Requirement already satisfied: pyeffects in
/home/vic/anaconda3/lib/python3.9/site-packages (1.0.5)
```

0.6 Dealing with Emptiness: Option


Nullable value of type A
Option[A]
implementations
Some[A](a: A)
a
The presence of a nullable value a of type A
None
The absence of a nullable value

0.6.1 An Example Data Model

[6]:
```python
from dataclasses import dataclass
from pyeffects.Option import *

@dataclass
class Name:
    first_name: str = None
    last_name: str = None

    def get_last_name(self) -> Option[str]:
        return Option.of(self.last_name)

    def get_first_name(self) -> Option[str]:
        return Option.of(self.first_name)

@dataclass
class Contact:
    name: Name = None
    relationship: str = "primary contact"

    def get_name(self) -> Option[Name]:
        return Option.of(self.name)

@dataclass
class Person:
    name: Name = None
    contact1: Contact = None
    contact2: Contact = None

    def get_name(self) -> Option[Name]:
        return Option.of(self.name)

    def get_contact1(self) -> Option[Contact]:
        return Option.of(self.contact1)

    def get_contact2(self) -> Option[Contact]:
        return Option.of(self.contact2)
```

[7]:
```
def get_contact_first_name(person: Person) -> str:
    if person and person.contact1 and person.contact1.name:
        return person.contact1.name.first_name
    else:
        return None

p = Person("Person 1", Contact(Name("Bob", "Smith"), "grandfather"),␣
 ↪Contact(Name("Mary", "Smith"), "mother"))

get_contact_first_name(p)
```

[7]:
```
'Bob'
```

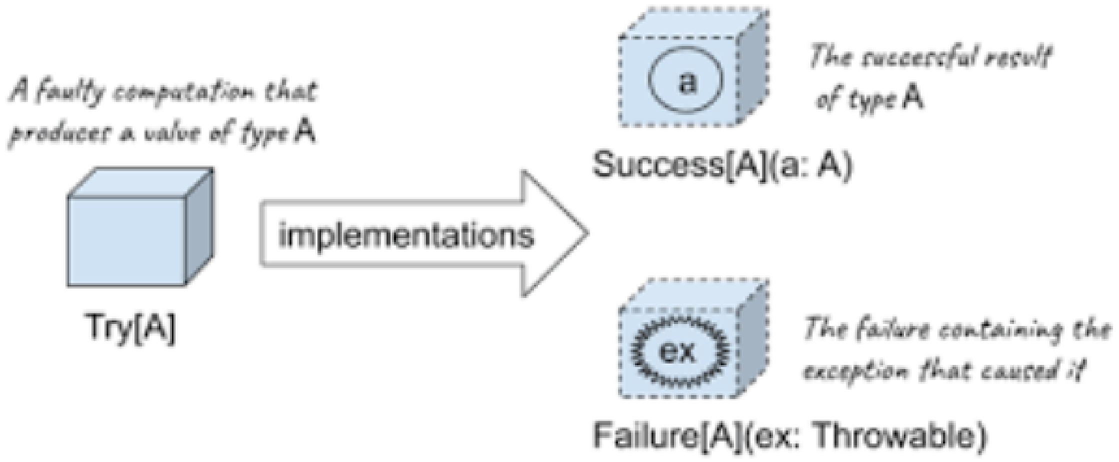
[8]:
```
def get_contact_first_name2(person: Person) -> Option[str]:
    return person.get_contact1()\
        .flat_map(Contact.get_name)\
        .flat_map(Name.get_first_name)

get_contact_first_name2(p)
```

[8]:
```
Some(Bob)
```

[9]:
```
person_with_no_contacts = Person("Person 2")
get_contact_first_name2(person_with_no_contacts).get_or_else("No contact name")
```

[9]:
```
'No contact name'
```

0.7 Dealing with Exceptions: Try


A faulty computation that produces a value of type A
Try[A]
implementations
The successful result of type A
a
Success[A](a: A)
The failure containing the exception that caused it
ex
Failure[A](ex: Throwable)

[10]:
```
person1_str = """
{
  "name": {
      "first_name": "1st",
      "last_name": "Person"
  },
  "contact1": {
      "name": {
          "first_name": "Alice",
          "last_name": "Jones"
      }
  }
}
"""

person2_str = """
{
  "name": {
      "first_name": "2nd",
      "last_name": "Person"
  },
  "contact1": {
      "name": {
          "first_name_is_wrong": "Mary",
          "last_name": "Jones"
      }
  }
}
"""
```

```
[11]: import json

      def parse_name(name_dict: dict) -> Name:
          try:
              n = Name(**name_dict)
              return n
          except TypeError as te:
              raise TypeError("Couldn't deserialize Name: " + str(te))

      def parse_contact(contact_dict: dict, relationship: str) -> Contact:
          n = None
          if 'name' in contact_dict:
              n = parse_name(contact_dict['name'])
          return Contact(name = n, relationship = relationship)

      def parse_person(json_str: str) -> Person:
          attributes = json.loads(json_str)
          n, contact1, contact2 = (None, None, None)
          if 'name' in attributes:
              n = parse_name(attributes['name'])
          if 'contact1' in attributes:
              contact1 = parse_contact(attributes['contact1'], 'contact1')
          if 'contact2' in attributes:
              contact2 = parse_contact(attributes['contact2'], 'contact2')
          return Person(n, contact1, contact2)

      person1 = parse_person(person1_str)
      person1.contact1.name.first_name
```

```
[11]: 'Alice'
```

```
[12]: person2 = parse_person(person2_str)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1818/2680335701.py in parse_name(name_dict)
      4     try:
----> 5         n = Name(**name_dict)
      6         return n

TypeError: __init__() got an unexpected keyword argument 'first_name_is_wrong'

During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1818/1474534612.py in <module>
----> 1 person2 = parse_person(person2_str)
```

```
/tmp/ipykernel_1818/2680335701.py in parse_person(json_str)
     20         n = parse_name(attributes['name'])
     21     if 'contact1' in attributes:
---> 22         contact1 = parse_contact(attributes['contact1'], 'contact1')
     23     if 'contact2' in attributes:
     24         contact2 = parse_contact(attributes['contact2'], 'contact2')

/tmp/ipykernel_1818/2680335701.py in parse_contact(contact_dict, relationship)
     11     n = None
     12     if 'name' in contact_dict:
---> 13         n = parse_name(contact_dict['name'])
     14     return Contact(name = n, relationship = relationship)
     15

/tmp/ipykernel_1818/2680335701.py in parse_name(name_dict)
      6         return n
      7     except TypeError as te:
----> 8         raise TypeError("Couldn't deserialize Name: " + str(te))
      9
     10 def parse_contact(contact_dict: dict, relationship: str) -> Contact:

TypeError: Couldn't deserialize Name: __init__() got an unexpected keyword␣
 ↪argument 'first_name_is_wrong'
```

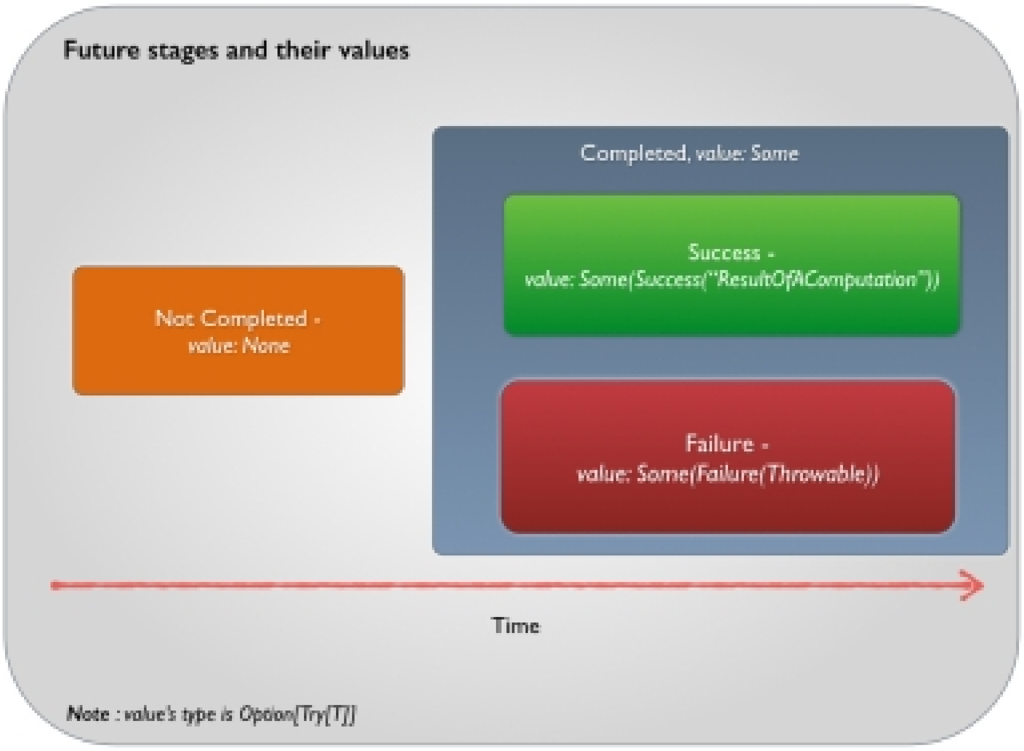
[13]:
```python
import json
from pyeffects.Try import *

def parse_name(name_dict: dict) -> Try[Person]:
    return Try.of(lambda: Name(**name_dict['name']))

def handle_parse_error(relationship: str) -> Contact:
    print(f"Error parsing contact: {relationship} (relationship)")
    return Contact(relationship = relationship)

def parse_contact(contact_dict: dict, relationship: str) -> Contact:
    return parse_name(contact_dict)\
        .map(lambda n: Contact(name = n, relationship = relationship))\
        .or_else_supply(lambda: handle_parse_error(relationship))

def parse_person(json_str: str) -> Person:
    attributes = json.loads(json_str)
    n = parse_name(attributes).get_or_else(None)
    contact1 = Try.of(lambda: parse_contact(attributes['contact1'],␣
 ↪'contact1')).get_or_else(None)
    contact2 = Try.of(lambda: parse_contact(attributes['contact2'],␣
 ↪'contact2')).get_or_else(None)
```

```
    return Person(n, contact1, contact2)
```

[14]:
```
person1 = parse_person(person1_str)
print(f"{person1.name.first_name} {person1.name.last_name}")
print(person1.contact1.name.first_name)

person2 = parse_person(person2_str)
print(f"{person2.name.first_name} {person2.name.last_name}")
print(person2.contact1.name)
```

```
1st Person
Alice
Error parsing contact: contact1 (relationship)
2nd Person
None
```

0.8 Dealing with Concurrency: Future


Future stages and their values
Completed, value: Some
Success -
value: Some(Success("ResultOfAComputation"))
Not Completed -
value: None
Failure -
value: Some(Failure(Throwable))
Time
Note : value's type is Option[Try[T]]

0.8.1 Running in Function a New Thread

[15]:
```
from pyeffects.Future import *
import time

def delayed_result() -> int:
    time.sleep(3)
    return 100

start_time = time.time()
result = Future.run(delayed_result).map(lambda v: v + 1)
print(f"Done: {result.is_done()}")
time.sleep(4)
print(f"Done: {result.is_done()}")
print(f"Result: {result.get()}")
execution_time = (time.time() - start_time)
print(f"Execution time: {execution_time} s")
```

```
Done: False
Done: True
Result: 101
Execution time: 4.003849029541016 s
```

0.8.2 Combining Futures using `flat_map`

[16]:
```
def delayed_result1() -> int:
    time.sleep(3)
    return 100

def delayed_result2() -> int:
    time.sleep(5)
    return 50

def handle_result(result: Try[int], start_time: int) -> None:
    print(f"Result: {result.get()}")
    execution_time = (time.time() - start_time)
    print(f"Execution time: {execution_time} s")


start_time = time.time()
result1 = Future.run(delayed_result1)
result2 = Future.run(delayed_result2)
result1.flat_map(lambda v1: result2.map(lambda v2: v1 + v2))\
    .on_complete(lambda v: handle_result(v, start_time))
```

```
Result: 150
```

```
Execution time: 5.007669448852539 s
```

0.8.3 Dealing with an Asynchronous Errors

[17]:
```
def delayed_result1() -> int:
    time.sleep(3)
    raise Exception("Error getting result")

def delayed_result2() -> int:
    time.sleep(5)
    return 50

def handle_result(result: Try[int], start_time: int) -> None:
    if result.is_failure():
        print("Unable to compute result due to exception")
        print(result.error())
    else:
        print(f"Result: {result.get()}")
    execution_time = (time.time() - start_time)
    print(f"Execution time: {execution_time} s")


start_time = time.time()
result1 = Future.run(delayed_result1)
result2 = Future.run(delayed_result2)
result1.flat_map(lambda v1: result2.map(lambda v2: v1 + v2))\
    .on_complete(lambda v: handle_result(v, start_time))
```

```
Unable to compute result due to exception
Error getting result
Execution time: 3.003845453262329 s
```

0.8.4 Using Future.traverse

[18]:
```
def delayed_result1() -> List[int]:
    time.sleep(3)
    return 100

def delayed_result2() -> List[int]:
    time.sleep(5)
    return 50

def handle_result(results: List[int], start_time: int) -> None:
    print(f"Result: {sum(results)}")
    execution_time = (time.time() - start_time)
    print(f"Execution time: {execution_time} s")
```

```python
start_time = time.time()
result1 = Future.run(delayed_result1)
result2 = Future.run(delayed_result2)

# Convert List[Future[int]] -> Future[List[int]]
Future.traverse([result1, result2])\
    .on_complete(lambda v: handle_result(v.get(), start_time))
```

```
Result: 150
Execution time: 5.003130674362183 s
```

0.9 Take aways

- Using side-effects can make code harder to reason about
- If functions are referentially transparent, it becomes easier to use localized reasoning
- Can use abstractions: Try, Future, Option, Either to replace common behaviors
- Fluent API style: reads left to right

0.9.1 Twitter: @vickumar1981

0.9.2 Github: https://github.com/vickumar1981