Język skryptowy lab4

Programowanie funkcyjne

Programowanie funkcyjne to styl programowania, który opiera się na funkcjach. Kluczową częścią programowania funkcjonalnego są funkcje wyższego rzędu. Funkcje wyższego rzędu przyjmują inne funkcje jako argumenty lub zwracają je jako wyniki.

```
def apply_twice(func, arg):
    return func(func(arg))

def add_five(x):
    return x + 5

print(apply_twice(add_five, 10))
```

> Funkcja **apply_twice** przyjmuje inną funkcję jako argument i wywołuje ją dwukrotnie wewnątrz.

Czyste funkcje

Programowanie funkcyjne ma na celu wykorzystanie czystych funkcji. czyste funkcje nie mają skutków ubocznych i zwracają wartość, która zależy tylko od ich argumentów. tak działają funkcje matematyczne: dla przykładu `cos (x)` dla tej samej wartości `x` , zawsze zwracają ten sam wynik poniżej są przykłady funkcji czystych i nieczystych

czysta funkcja

```
def pure_function(x, y):
    temp = x + 2 * y
    return temp / (2 * x + y)
```

nieczysta funkcja

```
some_list = []

def impure(arg):
    some_list.append(arg)
```

> Powyższa funkcja nie jest funkcją czystą ponieważ zmienia ona stan listy `some_list`

Używanie czystych funkcji ma zarówno zalety, jak i wady. Czyste funkcje są:

- bardziej zrozumiałe;

- łatwiejsze w testowaniu;
- bardziej wydajne;
- łatwiejsze do uruchomienia równolegle.

Wyrażenia Lambda

Tworzenie funkcji normalnie (za pomocą `def`) przypisuje ją do zmiennej automatycznie. Różni się to od tworzenia innych obiektów (takich jak łańcuchy i liczby całkowite) które można tworzyć w locie, bez przypisywania ich do zmiennej. To samo jest możliwe w przypadku funkcji, pod warunkiem, że są one tworzone przy użyciu wyrażenia **lambda**. Funkcje utworzone w ten sposób są znane jako anonimowe. To podejście jest najczęściej używane podczas przekazywania prostej funkcji jako argumentu do innej funkcji. Składnia jest pokazana w następnym przykładzie i składa się ze słowa kluczowego `lambda`, po którym następuje lista argumentów, dwukropek i wyrażenie do wyznaczenia i do zwrotu.

```
def my_func(f, arg):
    return (f(arg))

print(my_func(lambda x: 2 * x * x, 5))
```

Funkcje lambda nie są tak potężne, jak nazwane funkcje. Mogą tylko rzeczy, które wymagają pojedynczego wyrażenia - zwykle równoważne pojedynczemu wierszowi kodu.

```
# nzawana funkcja
def polynomial(x):
    return x ** 2 + 5 * x + 4

print(polynomial(-4))

# lambda
print((lambda x: x ** 2 + 5 * x + 4)(-4))
```

Funkcje lambda mogą być przypisane do zmiennych i używane jak normalne funkcje.

```
double = lambda x: x * 2
print(double(7))
```

> Jednak rzadko istnieje ku temu dobry powód - zazwyczaj lepiej jest zdefiniować funkcję z `def`

Funkcje `map` i `filter`

Wbudowane funkcje `map` i `filter` są bardzo przydatnymi funkcjami wyższego rzędu, które działają na listach (lub innych "iterowalnych" obiektach). Funkcja `map` przyjmuje funkcję i jest iterowalna jako argumenty i zwraca nową iterowalną z funkcją zastosowaną do każdego argumentu.

```
def add_five(x):
    return x + 5

nums = [11, 22, 33, 44, 55]
result = list(map(add_five, nums))
print(result)
```

Moglibyśmy osiągnąć ten sam wynik łatwiej dzięki składni lambda

```
nums = [11, 22, 33, 44, 55]

result = list(map(lambda x: x + 5, nums))
print(result)
```

> Aby przekonwertować wynik na listę, użyliśmy wyrażenia list

Funkcja `filter` jest iterowalna przez usunięcie elementów, które nie pasują do predykatu (funkcja zwracająca wartość logiczną).

```
nums = [11, 22, 33, 44, 55]
res = list(filter(lambda x: x % 2 == 0, nums))
print(res)
```

> Podobnie jak `map` , wynik musi zostać jawnie przekonwertowany na listę, jeśli chcesz ją wydrukować

Generatory

Generatory są typem iterowalnym jak listy lub krotki. W przeciwieństwie do list nie pozwalają na indeksowanie z dowolnymi indeksami, ale nadal mogą być iterowane za pomocą pętli `for` . Można je utworzyć za pomocą funkcji i instrukcji `yield`

```
def countdown():
    i = 5
    while i > 0:
        yield i
        i -= 1

for i in countdown():
    print(i)
```

> Instrukcja `yield` służy do zdefiniowania generatora, zastępując powrót funkcji, aby dostarczyć wynik do wywołującego bez niszczenia lokalnych zmiennych.

Ze względu na to, że przetwarzają po jednym elemencie, generatory nie mają ograniczeń dotyczących list pamięci. W rzeczywistości mogą być nieskończone!

```
def infinite_sevens():
    while True:
        yield 7

for i in infinite_sevens():
    print(i)
```

> W skrócie, generatory pozwalają zadeklarować funkcję, która zachowuje się jak iterator, tzn. może być użyta w pętli `for`

Generatory skończone można konwertować na listy, przekazując je jako argumenty funkcji listy

```
def numbers(x):
    for i in range(x):
        if i % 2 == 0:
            yield i

print(list(numbers(11)))
```

> Używanie generatorów prowadzi do zwiększenia wydajności, co jest wynikiem leniwego (na żądanie) generowania wartości, co przekłada się na mniejsze zużycie pamięci. Ponadto nie musimy czekać, aż wszystkie elementy zostaną wygenerowane, zanim zaczniemy z nich korzystać

Dekorator

Dekoratory umożliwiają modyfikowanie funkcji za pomocą innych funkcji. Jest to idealne rozwiązanie, gdy potrzebujesz rozszerzyć funkcjonalność funkcji, których nie chcesz modyfikować

```
def decor(func):
    def wrap():
        print("==================")
        func()
        print("==================")
    return wrap

def print_text():
    print("Hello world!")

decorated = decor(print_text)
decorated()
```

Zdefiniowaliśmy funkcję o nazwie `decor`, która ma jeden parametr `func`. Wewnątrz dekoru zdefiniowaliśmy zagnieżdżoną funkcję o nazwie `wrap`. Funkcja `wrap` wypisuje ciąg znaków, wywołuje

`func()` i wypisuje kolejny ciąg znaków. Funkcja `decor` zwraca funkcję `wrap` jako jej wynik. Można powiedzieć, że zmienna `decorated` to dekorowana wersja `print_text` - to `print_text` plus coś. W rezultacie, jeśli napiszemy użytecznego dekoratora moglibyśmy chcieć zastąpić `print_text` całkowicie dekorowaną wersją, więc zawsze mamy naszą wersję `print_text` wzbogaconą o coś nowego.

Dostawcy Python wspierają opakowywanie funkcji w dekoratorze poprzez oczekiwanie na definicję funkcji za pomocą nazwy dekoratora i symbolu @. Jeśli definiujemy funkcję, możemy ją ozdobić za pomocą symbolu @

```
@decor
def print_text2():
    print("Spam")

print_text2()
```

Rekurencja

Jest to bardzo ważna koncepcja w programowaniu funkcyjnym. Podstawą rekurencji jest samoodniesienie przez wywoływanie siebie. klasycznym przykładem rekurencji jest wyznaczanie silni.

```
def factorial(x):
    if x ==1:
        return 1
    else:
        return x*factorial(x-1)

print(factorial(5))
```

Rekurencja może być też pośrednia. Jedna funkcja wywołuje drugą która wywołuje pierwszą itd. Może się to zdarzyć w przypadku dowolnej liczby funkcji.

```
def is_even(x):
    if x == 0:
        return True
    else:
        return is_odd(x - 1)

def is_odd(x):
    return not is_even(x)

print(is_even(23))
print(is_odd(17))
```

Zbiory

Zbiory są strukturami danych podobnymi do list lub słowników. Są tworzone przy użyciu nawiasów klamrowych lub przy pomocy funkcji `set`. Mają one pewne funkcjonalności jak listy, tj. sprawdzanie czy zawierają konkretny przedmiot.

```
num_set = {1, 2, 3, 4, 5}
world_set = set(["spam", "eggs"])

print(3 in num_set)
print("spam" not in world_set)
```

Zbiory różnią się od list na kilka sposobów, ale umożliwiają kilka operacji dozwolonych na listach, takich jak `len`. Są nieuporządkowane, co oznacza, że nie można ich indeksować. Nie mogą zawierać duplikatów elementów. Ze względu na sposób ich przechowywania sprawdzenie czy element jest częścią zbioru, jest szybsze od sprawdzenia czy jest on częścią listy. Zamiast użycia `append` do dodania do zbioru używa się metody `add`. Metoda `remove` usuwa określony element ze zbioru, `pop` usuwa dowolny element.

```
nums = {1, 2, 1, 3, 4, 5, 1}
print(nums)
nums.add(-7)
nums.remove(3)
print(nums)
```

> Podstawowym zastosowaniem zbiorów jest testowanie występowania elementów i eliminację duplikatów

Zbiory można łączyć używając operatorów matematycznych:

- `|` suma zbiorów
- `&` część wspólna zbiorów
- `-` różnica zbiorów
- `^` różnica symetryczna

Moduł `itertools`

Znajduje się w standardowej bibliotece Pythona i posiada pewne użyteczne funkcje w programowaniu funkcyjnym.

Wiele funkcji w module itertools które działają na iteracjach podobnie do `map` i `filter':

- `takewhile` - pobiera element z iteracji jeśli funkcja predykatu jest prawdziwa
- `accumulate` - zwraca bieżącą sumę wartości w iteracji

```
nums = list(accumulate(range(9)))
print(nums)
```

```
print(list(takewhile(lambda x: x <= 6, nums)))
```

W module `itertools` jest również wiele funkcji kombinatorycznych tj. `product` i `permutations`. Są one używane, gdy chcemy wykonać zadanie ze wszystkimi możliwymi kombinacjami niektórych elementów.

```
letters = ("A", "B", "C")
print(list(product(letters, range(3))))
print(list(permutations(letters)))
```

Zadania do wykonania

:one: Wypróbuj kod z *listingów* znajdujących się w instrukcji i sprawdź ich działanie.

:two: Napisz funkcję z wykorzystaniem wyrażenia *lambda* która dla przyjętego argumentu `x` wyznaczy listę kwadratów kolejnych liczb naturalnych od `1` do `x`

:three: Z podanej poniżej listy odfiltruj elementy które są większe od 4 wykorzystując funkcję `filter`.

```
[1, 2, 3, 5, 8, 3, 0, 7]
```

:four: Napisz funkcję z wykorzystaniem generatora która wygeneruje poniższe elementy listy

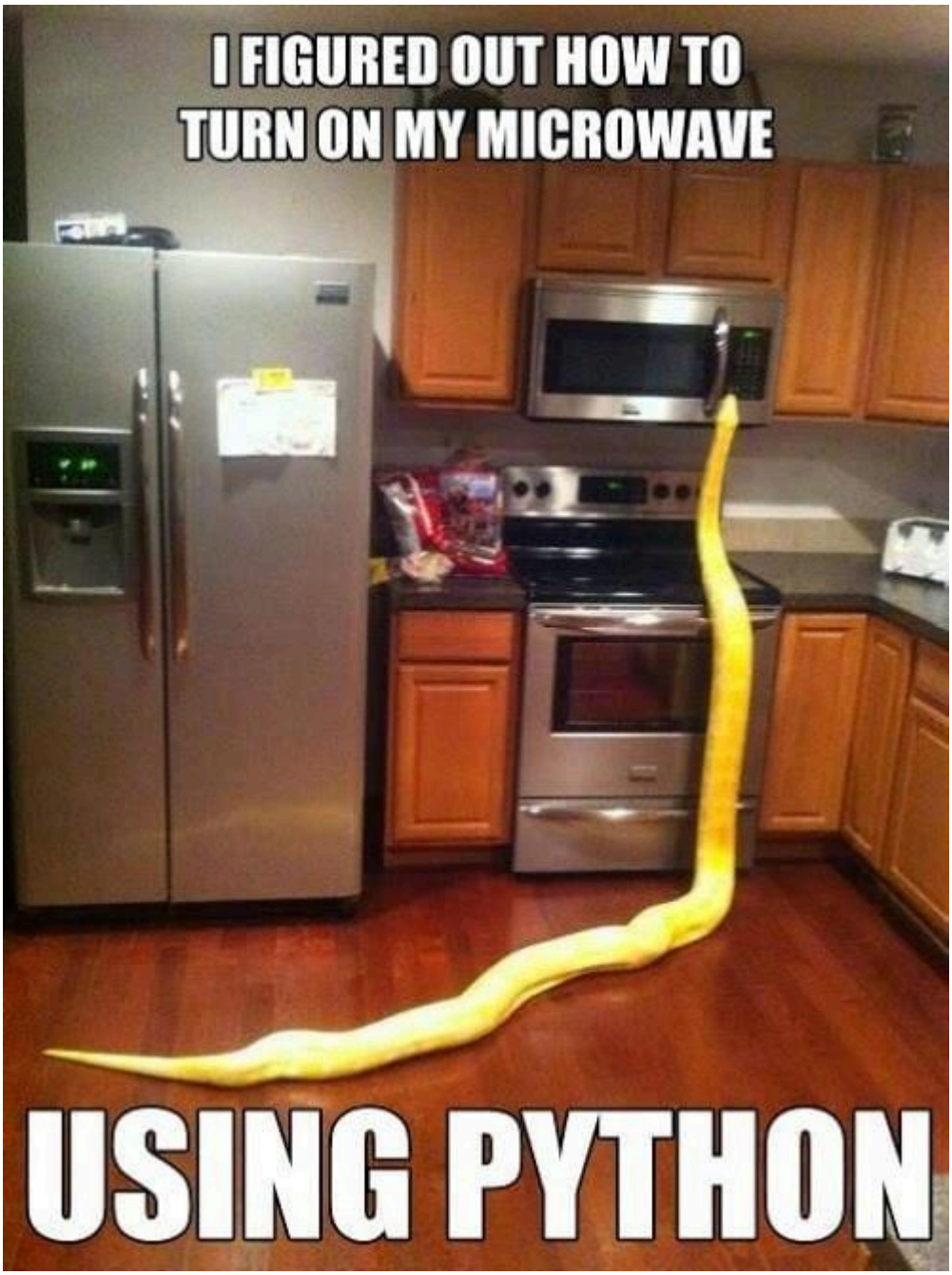
```
['s', 'sp', 'spa', 'spam']
```

:five: Napisz funkcję obliczającą dowolne równanie a następnie napisz funkcję dekorującą która przed wynikiem wstawi nazwę wywoływanej funkcji z jej parametrami wejściowymi.

:six: Napisz rekurencyjną funkcję przyjmującą 2 parametry która z podanych wartości obliczy wartość symbolu Newtona.

:seven: Napisz funkcję w której przetestujesz wszystkie omawiane operacje na zbiorach.

:exclamation: zadania 2-7 mają zostać dodane na GitHuba :exclamation:


I FIGURED OUT HOW TO
TURN ON MY MICROWAVE
USING PYTHON