2024-10-30

Элементы декларативного программирования в Python

Декларативное программирование — парадигма, при которой определяется ожидаемый результат выполнения программы, а не последовательность отдельных конкретных шагов, которые ведут к нему (в отличие от императивной парадигмы).

Функциональное программирование — подвид декларативного, при котором процесс выполнения программы интерпретируется концептуально как вычисление значений функций (с позиции математического определения функции)

К функциональным ЯП традиционно относят:

- LISP
- F#
- Erlang -> Elixir
- Haskell
- Scala
- *Эксель-функции

Элементы функциональной парадигмы присутствуют во многих ЯП, включая Python.

```
(+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```

```
lambda n:reduce(lambda x,n:[x[1],x[0]+x[1]], range(n),[0,1])[0]
```

Включения (comprehensions)

Списковые и словарные включения как часть более общей категории выражений-генераторов (PEP 289 – Generator Expressions | peps.python.org)

```
list_comp = [letter*2 for letter in 'abcdefg']
dict_comp = {f'id_{x}': x**2 for x in range(1000)}
```

Генераторы (generators)

Часть более общей категории, описанной в PEP-289

Однострочный генератор:

```
single_line_gen = (letter*2 for letter in 'abcdefg')
```

Многострочные генераторы:

```
for prog_idx, prog in enumerate(set_1):
    for mach_idx, mach in enumerate(prog):
        for work_idx, work in enumerate(mach):
            yield (prog_idx, mach_idx, work_idx)
```

Генераторы не хранят в памяти все свои возможные значения, а только текущее значение и правила вывода следующего.

Такой подход обеспечивает экономию памяти (и поэтому в Питоне 3 возможно `range(100_000_000_000_000_000_000_000)`), но генератор при этом работает только в 1 сторону. Чтобы вернуться к уже пройденным значениям, генератор необходимо пересоздать.

Следующее значение получается за счет передачи генератора во встроенный метод `next()` или в ходе итерации по аналогии с традиционными итерируемыми коллекциями (но такая итерация возможна лишь 1 раз).

При исчерпании генератора (правило не позволяет получить следующее значение), т.е. при вызове `next()` на генератор с максимальным допустимым значением или при попытке повторной итерации по генератору, будет выброшено исключение `StopIteration`.

Анонимные функции (лямбда-функции, lambda)

```
import math

lambda_sqrt = lambda x: math.sqrt(x+1 if x > 0 else x)

lambda_sqrt(50)
lambda_sqrt(100)
```

```
[skill in map(lambda x: x[0].lower(), cluster) for skill in
context_skills]
```

Функции высшего порядка (Higher-Order Functions, HOF)

Функция высшего порядка — функция, которая принимает в качестве аргумента другую функцию и/или возвращает функцию как результат своего вычисления.

А т.к. в Питоне любая функция всегда является одновременно объектом, никто не запрещает передавать этот объект в качестве аргумента в другие функции аналогично другим переменным или возвращать его при вызове функции.

```
import sys

def display(fun, arg):
    print(f'{type(fun)} : {fun}')
    print(f'arg={arg} => fun(arg)={fun(arg)}'

if len(sys.argv) > 1:
    n = float(sys.argv[ 1 ])
else:
    n = float( input( "число?: " ) )

def pow3(n):  # 1-е определение функции
    return n * n * n
display(pow3, n)

pow3 = lambda n: n * n * n # 2-е определение функции
display(pow3, n)

display((lambda n: n * n * n), n)  # 3-е определение функции, ad-hoc
lambda
```

```
'''
[out]
<class 'function'> : <function pow3 at 0xb74542ac>
arg=1.3 => fun( arg )=2.1970000000000005
<class 'function'> : <function <lambda> at 0xb745432c>
arg=1.3 => fun( arg )=2.1970000000000005
<class 'function'> : <function <lambda> at 0xb74542ec>
arg=1.3 => fun( arg )=2.1970000000000005
'''
```

Декораторы

Синтаксический сахар для модификации функций другой функцией высшего порядка

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")

# Output: I am ordinary

@make_pretty
def ordinary():
    print("I am ordinary")

# I got decorated
# I am ordinary

decorated_func = make_pretty(ordinary)
```

map()

Применяет переданную функцию к каждому элементу в переданной последовательности и возвращает последовательность результатов той же размерности, что и входная.

```
[skill in map(lambda x: x[0].lower(), cluster) for skill in
context_skills]
```

reduce()

Применяет переданную функцию к каждому значению в списке и ко внутреннему накопителю результата (на примере обозначен как `aggr`) — агрегатору.

Пример: вычисление факториала 10!

```
from functools import reduce

reduce(lambda aggr, m: aggr*m, range(1, 11))
```

```
from functools import reduce

reduce(lambda aggr, b: aggr+b[1][0], [skill for skill in cluster if
skill[1][1].lower() in context_skills], 0.0)
```

В JavaScript оно даже более наглядно:

```
console.log([1, 2, 3, 4, 5].reduce((a, b) => {return a+b}, 0)) // на
выходе 15
```

filter()

Применяет переданную функцию к каждому элементу коллекции и возвращает коллекцию тех элементов исходной коллекции, для которых переданная функция вернула значение истинности.

```
list(filter(lambda x: len(x) >= 3, ready_clusters))
```

> **Note**
>
> Функции `map()` и `filter()` возвращают не список, а особые объекты-итераторы

Но при этом `filter()` всегда можно заменить генератором:

```
(x for x in ready_clusters if len(x) >= 3)
```