Парадигмы программирования

Для начала обратимся к википедии за определением понятия “парадигма программирования”. **Парадигма программирования** — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Выделяют две крупные парадигмы программирования: **императивная** и **декларативная**.

Императивное программирование предполагает ответ на вопрос “Как?”. В рамках этой парадигмы вы задаете последовательность действий, которые нужно выполнить, для того чтобы получить результат. Результат выполнения сохраняется в ячейках памяти, к которым можно обратиться в последствии.

Декларативное программирование предполагает ответ на вопрос “Что?”. Здесь вы описываете задачу, даете спецификацию, говорите, что вы хотите получить в результате выполнения программы, но не определяете, как этот ответ будет получен.

Каждая из этих парадигм включает в себя более специфические модели. В промышленности наибольшее распространение получили **структурное** и **объектно-ориентированное программирование** из группы “императивное программирование” и **функциональное программирование** из группы “декларативное программирование”.

В рамках **структурного** подхода к программированию основное внимание сосредоточено на декомпозиции – разбиении программы/задачи на отдельные блоки / подзадачи. Разработка ведётся пошагово, методом “сверху вниз”. Наиболее распространенным языком, который предполагает использование структурного подхода к программирования является язык *C*, в нем, основными строительными блоками являются функции.

В рамках **объектно-ориентированного (ООП)** подхода программа представляется в виде совокупности объектов, каждый из которых является экземпляром определенного класса, классы образуют иерархию наследования. ООП базируется на следующих принципах: инкапсуляция, наследование, полиморфизм, абстракция. Примерами языков, которые позволяют вести разработку в этой парадигме являются *C#*, *Java*.

В рамках **функционального** программирования выполнение программы – это процесс вычисления, который трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Языки, которые реализуют эту парадигму – это *Haskell*, *Lisp*.

Довольно часто бывает так, что дизайн языка позволяет использовать все перечисленные парадигмы (например *Python*) или только часть из них (например, *C++*).

В чем суть функционального программирования?

Языки, которые можно отнести в функциональной парадигме обладают определенным набором свойств. Если язык не является чисто функциональным, но реализует эти свойства, то на нем можно разрабатывать, как говорят, в функциональном стиле. Свойства функционального стиля программирования:

- Функции являются объектами первого класса (*First Class Object*). Это означает, что с функциями вы можете работать, также как и с данными: передавать их в качестве аргументов другим функциям, присваивать переменным и т.п.
- В функциональных языках не используются переменные (как именованные ячейки памяти), т.к. там нет состояний, а т.к. нет переменных, то и нет операции присваивания, как это понимается в императивном программировании.
- Рекурсия является основным подходом для управления вычислениями, а не циклы и условные операторы.
- Используются функции высшего порядка (*High Order Functions*). Функции высшего порядка – это функций, которые могут в качестве аргументов принимать другие функции.
- Функции являются “чистыми” (*Pure Functions*) – т.е. не имеют побочных эффектов (иногда говорят: не имеют сайд-эффектов).
- Акцент на том, что должно быть вычислено, а не на том, как вычислять.
- Фокус на работе с контейнерами (хотя это спорное положение).

Python не является функциональным языком программирования, но его возможностей хватает, чтобы разрабатывать программы в функциональном стиле

Функциональный код отличается одним свойством: отсутствием побочных эффектов. Он не полагается на данные вне текущей функции, и не меняет данные, находящиеся вне функции. Все остальные «свойства» можно вывести из этого.

Нефункциональная функция:

```
a = 0
def increment1():
    global a
    a += 1
```

Функциональная функция:

```
def increment2(a):
    return a + 1
```

Lambda выражение в Python:

lambda оператор или **lambda функция в Python** это способ создать анонимную функцию, то есть функцию без имени. Такие функции можно назвать одноразовыми, они используются только при создании. Как правило, **lambda функции** используются в комбинации с функциями filter, map, reduce.

Синтаксис lambda выражения в Python

```
lambda arguments: expression
```

В качестве arguments передается список аргументов, разделенных запятой, после чего над переданными аргументами выполняется expression. Если присвоить lambda-функцию переменной, то получим поведение как в обычной функции (делаем мы это исключительно в целях демонстрации)

```
>>> multiply = lambda x,y: x * y
>>> multiply(21, 2)
42
```

Но, конечно же, все преимущества lambda-выражений мы получаем, используя lambda в связке с другими функциями

Функция map() в Python:

В **Python функция map** принимает два аргумента: функцию и аргумент составного типа данных, например, список. map применяет к каждому элементу списка переданную функцию. Например, вы прочитали из файла список чисел, изначально все эти числа имеют строковый тип данных, чтобы работать с ними - нужно превратить их в целое число:

```
old_list = ['1', '2', '3', '4', '5', '6', '7']

new_list = []
for item in old_list:
    new_list.append(int(item))

print (new_list)

[1, 2, 3, 4, 5, 6, 7]
```

Тот же эффект мы можем получить, применив функцию map:

```
old_list = ['1', '2', '3', '4', '5', '6', '7']
new_list = list(map(int, old_list))
print (new_list)

[1, 2, 3, 4, 5, 6, 7]
```

Как видите такой способ занимает меньше строк, более читабелен и выполняется быстрее. map также работает и с функциями созданными пользователем:

```
def miles_to_kilometers(num_miles):
    """ Converts miles to the kilometers """
    return num_miles * 1.6

```

```
mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(miles_to_kilometers, mile_distances))
print (kilometer_distances)

[1.6, 10.4, 27.84, 3.84, 14.4]
```

А теперь то же самое, только используя **lambda выражение**:

```
mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(lambda x: x * 1.6, mile_distances))

print (kilometer_distances)

[1.6, 10.4, 27.84, 3.84, 14.4]
```

Функция map может быть так же применена для нескольких списков, в таком случае функция-аргумент должна принимать количество аргументов, соответствующее количеству списков:

```
l1 = [1,2,3]
l2 = [4,5,6]

new_list = list(map(lambda x,y: x + y, l1, l2))
print (new_list)

[5, 7, 9]
```

Если же количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
l1 = [1,2,3]
l2 = [4,5]

new_list = list(map(lambda x,y:  + y, l1, l2))

print (new_list)
[5,7]
```

Функция filter() в Python:

Функция filter предлагает элегантный вариант фильтрации элементов последовательности. Принимает в качестве аргументов функцию и последовательность, которую необходимо отфильтровать:

```
mixed = ['мак', 'просо', 'мак', 'мак', 'просо', 'мак', 'просо', 'просо',
'просо', 'мак']
zolushka = list(filter(lambda x: x == 'мак', mixed))

print (zolushka)
 ['мак', 'мак', 'мак', 'мак', 'мак']
```

Обратите внимание, что функция, передаваемая в **filter** должна возвращать значение True / False, чтобы элементы корректно отфильтровались.

Функция reduce() в Python:

Функция reduce принимает 2 аргумента: функцию и последовательность. reduce() последовательно применяет функцию-аргумент к элементам списка, возвращает единичное значение. Обратите внимание в Python 2.x функция reduce доступна как встроенная, в то время, как в Python 3 она была перемещена в модуль functools.

Вычисление суммы всех элементов списка при помощи reduce:

```
?
1from functools import reduce
2items = [1,2,3,4,5]
3sum_all = reduce(lambda x,y: x + y, items)
4
5print (sum_all)
6
715
```

Вычисление наибольшего элемента в списке при помощи reduce:

```
?
1from functools import reduce
2items = [1, 24, 17, 14, 9, 32, 2]
3all_max = reduce(lambda a,b: a if (a > b) else b, items)
4
5print (all_max)
632
```

Функция zip() в Python:

Функция zip объединяет в кортежи элементы из последовательностей переданных в качестве аргументов.

```
?
1a = [1,2,3]
2b = "xyz"
3c = (None, True)
4
5res = list(zip(a, b, c))
6print (res)
7
8[(1, 'x', None), (2, 'y', True)]
```

Обратите внимание, что **zip** прекращает выполнение, как только достигнут конец самого короткого списка.