**Парадигмы программирования**

Для начала обратимся к [википедии](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) 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

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1 | **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)

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3 | >>> 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:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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](http://pythonicway.com/python-functions) созданными пользователем:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **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**:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **from** functools **import** reduce  items **=** [1,2,3,4,5]  sum\_all **=** reduce(**lambda** x,y: x **+** y, items)    print (sum\_all)    15 |

Вычисление наибольшего элемента в списке при помощи reduce:

[?](http://pythonicway.com/python-functinal-programming#)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **from** functools **import** reduce  items **=** [1, 24, 17, 14, 9, 32, 2]  all\_max **=** reduce(**lambda** a,b: a **if** (a > b) **else** b, items)    print (all\_max)  32 |

Функция zip() в Python:

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

[?](http://pythonicway.com/python-functinal-programming#)

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

Обратите внимание, что **zip** прекращает выполнение, как только достигнут конец самого короткого [списка](http://pythonicway.com/python-data-types#list).