

Оглавление

1	Тестирование и отладка программ	2
1.1	Знакомство с курсом	2
1.2	Структурное программирование	3
1.2.1	Профессионализм в программировании	3
1.2.2	Культура программирования	4
1.2.3	Выбор идентификаторов	5
1.2.4	Структурное программирование	6
1.2.5	Проектирование приложения «сверху вниз»	7
1.3	Тестирование и отладка	11
1.3.1	Зачем нужно тестировать программу	11
1.3.2	Контрактное программирование	14
1.3.3	Модульное тестирование и Test-Driven Development	16
1.3.4	Библиотека doctest	19
1.3.5	Библиотека unittest	21

Неделя 1

Тестирование и отладка программ

1.1 Знакомство с курсом

Структура курса:

1. Неделя 1

- Культура программирования;
- Проектирование «сверху-вниз»;
- Программирование по контракту;
- Тестирование.

2. Неделя 2

- Объектно-ориентированное мышление;
- UML-диаграммы.

3. Неделя 3

- Паттерны проектирования;
- «Декоратор», «Адаптер», «Наблюдатель».

4. Неделя 4

- Паттерн «Цепочка обязанностей»;
- Паттерн «Абстрактная фабрика»;
- Конфигурация программ.

5. Неделя 5

- Курсовой проект.

1.2 Структурное программирование

1.2.1 Профессионализм в программировании

Профессиональный программист отличается от любителя тем, что он:

- адекватно оценивает:
 - срок создания программного продукта;
 - необходимые для этого ресурсы.
- хорошо планирует:
 - архитектуру программного продукта;
 - последовательность разработки.
- выполняет работу качественно:
 - ПО надёжно функционирует;
 - программный код легко читать и сопровождать.

Классический процесс это каскадная модель разработки ПО (**Waterfall**):

1. Анализ требований к ПО
2. Разработка архитектуры программы
3. Кодирование
4. Тестирование и отладка
5. Инсталляция и поддержка

Проектирование архитектуры программы помогает понять, что именно надо делать и когда. Профессиональному программисту важно:

- Знать парадигмы программирования:
 - Структурная парадигма (движение «сверху-вниз»);
 - Модульная парадигма (разбиение программы на отдельные модули);
 - Объектно-ориентированное проектирование.
- Использовать готовые архитектурные решения - паттерны проектирования;
- Уметь визуализировать дизайн программы.

Читабельность кода (**readability**) очень важна, ведь она даёт возможность быстро понимать смысл исходного текста, лучше видеть ошибки, увереннее модифицировать код и уменьшить объём внутренней документации.

Важно и качество работы программы. Она должна делать то, что заявлено в техническом задании (ТЗ) без ошибок и быстро.

1.2.2 Культура программирования

Код читается намного больше раз, чем пишется и поэтому критически важна «читабельность» программного кода. Для этого в Python существует универсальный стиль кода **PEP8**:

1. Внешний вид кода:

- Кодировка для файла исходного текста только юникод т.е. **UTF-8**. А идентификаторы, переменные, функции и комментарии **ASCII**;
- Никогда не смешивать табуляции и пробелы;
- Желательно 4 пробела на один уровень отступа;
- Ограничить максимальную длину строки 79-ю символами (для этого можно использовать уже имеющиеся скобки или обратный слэш). Перенос строки делать строго после бинарного оператора.

2. Пробелы в выражениях и инструкциях

- Окружайте ровно одним пробелом с каждой стороны:
 - операторы присваивания (`=`, `+=`, `-=` и т.п.);
 - операторы сравнения (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`);
 - логические операторы (`and`, `or`, `not`).
- Ставьте пробелы вокруг арифметических операций
- Избегать пробелов:
 - Сразу после или перед скобками `()`, `[]`, `{}`;
 - Перед запятой, точкой с запятой, двоеточием;
 - Перед открывающейся скобкой при вызове функций или скобкой, после которой следует индекс или срез.

3. Пустые строки

- Используйте пустые строки, чтобы отделить друг от друга логические части функции;
- Отделять одной пустой строкой определения методов внутри класса;
- Функции верхнего уровня и определения классов отделять двумя пустыми строчками.

4. Комментарии

- Не объясняйте очевидное и обновляйте комментарии вместе с кодом;

- Блок комментариев обычно объясняет код, идущий *после* блока, и должен иметь тот же отступ, что и код;
- «Встрочные» комментарии отделяйте хотя бы двумя пробелами от инструкции и начинайте их с символа `#` и одного пробела;
- Там, где это возможно, вместо комментариев используйте документ-строки;
- Пишите комментарии на грамотном английском языке. Первое слово с большой буквы и в конце ставьте точку. Предложения отделяйте двумя пробелами.

1.2.3 Выбор идентификаторов

Соглашение об именах:

- Модуль (или пакет) должен называться коротко, записываться маленькими буквами и без подчёркиваний. Например, **`mymodule`**;
- Классы и исключения называются несколькими словами слитно, каждое из которых с большой буквы. Например, **`UserClassName`**;
- Функции, переменные и методы записываются несколькими словами с маленькой буквы, через знак подчёркивания. Например, **`function_name`**;
- Иногда функции называют так: **`notDesiredFunctionName`**, но первая буква - маленькая;
- Глобальные константы пишутся заглавными буквами через подчёркивание: **`GLOBAL_CONSTANT`**.

Использование символа подчёркивания:

- В начале — функция для внутренних нужд: **`_internal_function`**;
- В конце — избегание конфликта с зарезервированным словом: **`reserve_`**;
- Два подчёркивания в начале — скрываемая функция или атрибут: **`__hidden`**;
- Два в начале, два в конце — функция с особым использованием (согласно документации). **`__magic_method__`**.

Глобальные переменные:

- Это плохой стиль. Их надо избегать везде, где возможно;
- Поведение функций начинает зависеть от неявно заданных обстоятельств;
- Мешают распараллеливать код.

Рассмотрим пример использования **PEP8**:

```

class MyClass:
    """
    Use PEP8 to be a Master of Code!
    """
    def __init__(self, class_):
        self._internal = class_
    def public_method(self, x: int):
        """
        some document string here
        """
        if x > 0:
            return x + 1
    return x

```

Видим, что здесь соблюдаются все вышеперечисленные рекомендации.
Замечания:

- В случае конфликта стилей, если имя — часть **API**, то оно должно быть согласовано со стилем кода интерфейса, а не реализации;
- Имена должны содержать только символы ASCII и означать только английские слова;
- Имена должны отражать смысл объекта.

[Краткая версия PEP8 на русском](#)

[Полная версия PEP8 на английском языке](#)

1.2.4 Структурное программирование

Структурное программирование — это методология, облегчающая создание больших программ. У начинающих программистов проблема в том, что они сконцентрированы на синтаксисе языка. А методология это то, что помогает приподняться над синтаксисом. Рассмотрим аналогию с водителями:

Начинающий водитель думает о мелких технических моментах:

- Как не врезаться?
- Разрешён ли обгон?
- Можно ли тут останавливаться?

Опытный водитель задаёт глобальные вопросы:

- Какая цель у поездки?
- Каков оптимальный маршрут?
- Как объехать нужные места?

Аналогично, начинающий программист думает о синтаксисе и деталях.

Может быть, этот кусок программы вообще не надо писать.

Опытные программисты думают о безопасности, комфорте и надёжности.

Принципы структурного программирования:

- Откажитесь от использования `goto`;
- Стройте программу из вложенных конструкций: последовательность, ветвление, циклы;
- Оформляйте повторяющиеся фрагменты программы в виде функций;
- Разработка программы ведётся пошагово, методом «сверху вниз».

1.2.5 Проектирование приложения «сверху вниз»

Рассмотрим проектирование приложения «сверху вниз» на примере создания графического приложения с библиотекой графики. Импортируем библиотеку `graphics`, пишем и вызываем главную функцию, создающую главное окно и вызывающую функцию `draw_image` для отрисовки картинки:

```
import graphics as gr #импорт библиотеки

def main():
    window = gr.GraphWin("My Image", 600, 600)
    draw_image(window)
    window.getMouse()

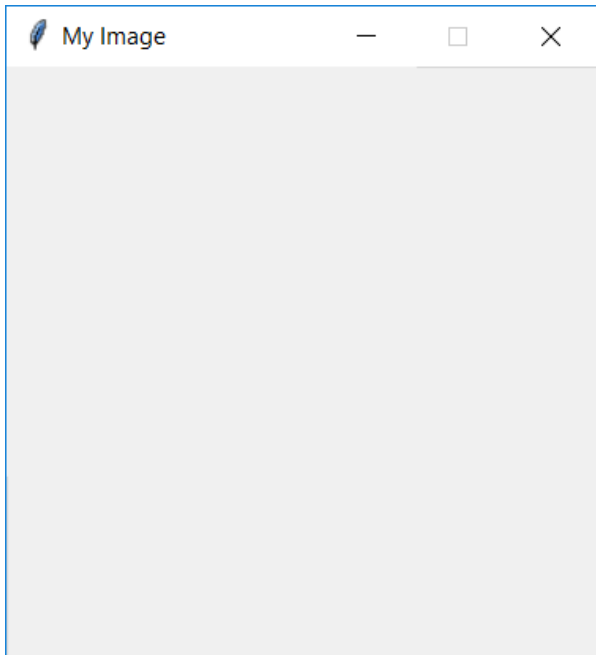
def draw_image(window):
    house_x, house_y = window.width // 2, \
                        window.height * 2 // 3
    house_width = window.width // 3
    house_height = house_width * 4 // 3

    # отделили рисующие функции от расчёта параметров для них
    draw_background(window)
    draw_house(window, house_x, house_y,
               house_width, house_height)

def draw_background(window):
    pass # FIXME

def draw_house(window, x, y, width, height):
    pass # FIXME

if __name__ == "__main__":
    main()
```

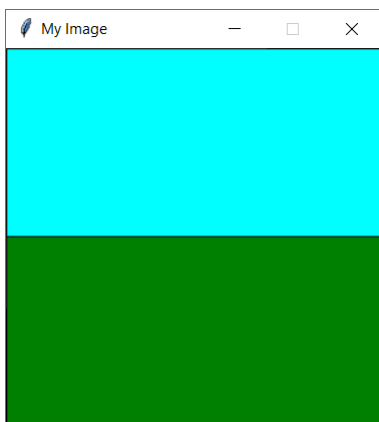


При запуске программы у нас появляется окошко, значит всё работает. Далее по тому же принципу (спускаясь от функций верхнего уровня к вызываемым) допишем код. Мы все время делегируем выполнение какой-то подзадачи в отдельную функцию, а сам процесс этого разбиения задачи на подзадачи называется **декомпозицией**.

Вставим рисование бэкграунда: земля на пол экрана и небо

```
def draw_background(window):
    earth = gr.Rectangle(gr.Point(0, window.height // 2),
                        gr.Point(window.width - 1,
                                window.height - 1))

    earth.setFill("green")
    earth.draw(window)
    sci = gr.Rectangle(gr.Point(0, 0),
                    gr.Point(window.width - 1, window.height // 2))
    sci.setFill("cyan")
    sci.draw(window)
```



Запустив программу видим, нарисованный фон, потому что эта функция уже вызвана. При движении снизу вверх мы бы вначале написали эту функцию, а потом бы придумывали, как ее использовать. Аналогия: можно делать велосипед, начиная с деталей, но не продумав, как их собирать. А мы делаем, спускаясь сверху вниз. Сначала вызвали функцию и поставили на неё заглушку, а потом, спустившись вниз, сделали бэкграунд чтобы он отвечал требованию — залить всё окно.

Пропишем draw_house: мы хотим дом из цоколя, стен и крыши.

```
def draw_house(window, x, y, width, height):
    foundation_height = height // 8
    walls_height = height // 2
    walls_width = 7 * width // 8
    roof_height = height - walls_height - \
        foundation_height

    # вызов самих функций рисования как будто они существуют
    draw_house_foundation(window, x, y, width,
                           foundation_height)
    draw_house_walls(window, x, y - foundation_height,
                     walls_width, walls_height)
    draw_house_roof(window, x,
                    y - foundation_height - walls_height,
                    width, roof_height)
```

Теперь функции рисования цоколя, стен и крыши:

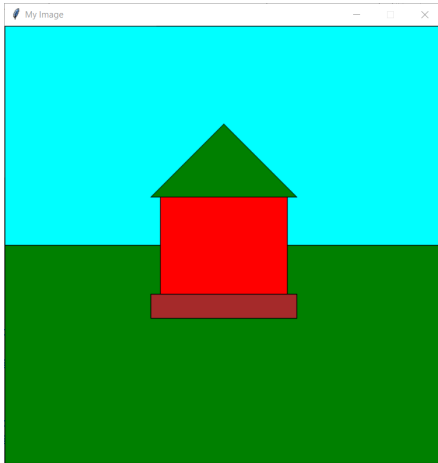
```
def draw_house_foundation(window, x, y, width, height):
    foundation = gr.Rectangle(gr.Point(x - width // 2, y),
                              gr.Point(x + width // 2,
                                         y - height))

    foundation.setFill("brown")
    foundation.draw(window)

def draw_house_walls(window, x, y, width, height):
    walls = gr.Rectangle(gr.Point(x - width // 2, y),
                          gr.Point(x + width // 2,
                                     y - height))

    walls.setFill("red")
    walls.draw(window)
    draw_house_window(window, x, y - height // 4,
                      width // 3, height // 2)

def draw_house_window(window, x, y, width, height):
    pass # FIXME
```



На окно пока поставили заглушку, но и без него домик отрисовывается. Теперь допишем `draw_house_window`:

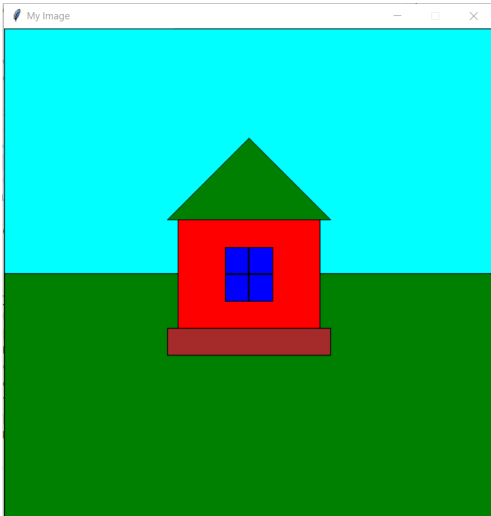
```
def draw_house_window(window, x, y, width, height):
    glass = gr.Rectangle(gr.Point(x - width // 2, y),
                        gr.Point(x + width // 2,
                                y - height))

    glass.setFill("blue")
    line1 = gr.Line(gr.Point(x, y),
                    gr.Point(x, y - height))
    line2 = gr.Line(gr.Point(x - width // 2,
                            y - height // 2),
                    gr.Point(x + width // 2,
                            y - height // 2))

    glass.draw(window)
    line1.draw(window)
    line2.draw(window)
    line1.setOutline("black")
    line2.setOutline("black")
    line1.setWidth(2)
    line2.setWidth(2)

def draw_house_roof(window, x, y, width, height):
    roof = gr.Polygon(gr.Point(x - width // 2, y),
                     gr.Point(x + width // 2, y),
                     gr.Point(x, y - height))

    roof.setFill("green")
    roof.draw(window)
```



Таким образом, мы получили готовый домик с окошком и крышей. Структурное программирование прекрасно ложится на групповую работу программистов, когда подзадача делегирована. И она не просто делегирована в отдельную функцию, она еще и передана конкретному программисту, который может рисовать окно как угодно, но не вылезая за x и y . Это очень удобно с точки зрения дальнейшей модификации и работы — если захочется нарисовать ещё три таких домика, можно просто из главной функции сделать пару вызовов с другими параметрами.

Итеративная разработка с проектированием «сверху вниз»

1.3 Тестирование и отладка

1.3.1 Зачем нужно тестировать программу

Обратная связь — это данные, которые с выхода поступают на вход. Её типы:

- Положительная обратная связь усиливает сигнал на выходе. В случае разработки это:
 - положительные отзывы конечных пользователей;
 - запросы пользователей на новую функциональность;
 - увеличение объема продаж.
- Отрицательная обратная связь гасит сигнал
 - негативные отзывы конечных пользователей;
 - отсутствие интереса к программному продукту;
 - падение объёма продаж.

К сожалению, отрицательная обратная связь обычно поступает слишком поздно. Тут-то и нужен **тестировщик** — человек, который даёт участникам проекта по разработке ПО отрицательную обратную связь о качестве программного продукта на самой ранней стадии, когда ещё не поздно всё исправить.

Тестирование (Quality Control) — это проверка соответствия между реальным и ожидаемым поведением программы, которая проводится на конечном наборе специально выбранных тестов.

Обязанности тестировщика:

- находить дефекты («баги»);
- вносить описание найденного дефекта в систему отслеживания ошибок (Bug tracking system);
- описывать способы воспроизведения ошибок;
- создавать отчёты о тестировании для каждой версии продукта;
- (дополнительно) читать и исправлять документацию;
- (дополнительно) анализировать и уточнять требования к программе;
- (дополнительно) создавать ПО для автоматизации процесса тестирования.

Главный результат работы тестировщика - повышение качества ПО.
Аспекты качества программы:

1. функциональность:

- пригодность к использованию;
- правильность выполнения задачи;
- поддержка стандартов;
- защищенность (security).

2. надёжность:

- низкая частота отказов;
- отказоустойчивость;
- способность к восстановлению;

3. практичность (user-friendly):

- понятность в использовании;
- управляемость;
- привлекательность;

4. эффективность:

- время отклика программы;
- объём использования ресурсов ПК;

5. сопровождаемость;

6. переносимость.

И для каждого аспекта качества программы есть соответствующий вид тестирования. Например, функциональное тестирование проверяет пригодность к использованию, правильность работы и защищенность программы.

Классификация тестирования по масштабу (как их видит тестировщик):

1. **м**одульное тестирование (unit testing) — тестирование отдельных операций, методов и функций;
2. интеграционное тестирование — проверка корректности взаимодействия модулей между собой;
3. системное тестирование — тестирование на уровне пользовательского интерфейса.

С технической точки зрения эти три вида тестирования похожи друг на друга: если у вас есть какой-то инструмент, например, модульного тестирования, то его иногда можно применить и к системному тестированию.

Крайне желательно иметь тестировщика у себя в команде, когда проект разрастётся, но кто может быть тестировщиком на начальных этапах?

- Программист или Старший программист (Team Leader) не могут быть тестировщиками. Они ходят только «по протоптанному», жалеют собственный код и не сильно придираются к нему;
- Менеджер проекта, технический писатель, эксперт предметной области или представитель заказчика могут, потому что тестировщик должен не знать деталей реализации, воспринимать программу как чёрный ящик и не быть слишком привязанным к ней;
- Методика **test-driven development (TDD)** позволяет программисту быть самому себе эффективным тестировщиком.

Когда ошибка найдена (самостоятельно или с помощью тестировщика), необходимо **отладить** программу:

1. Понять суть ошибки и обнаружить её причину;
2. Локализовать её в исходном тексте программы;
3. Устранить ошибку.

Типичные задачи отладки: узнать текущие значения переменных и выяснить, по какому пути выполнялась программа.

Существуют два пути отладки:

- Использование отладчиков («дебаггеров»);
- **л**оггирование (вывод отладочных сведений в файл).

1.3.2 Контрактное программирование

Поиск ошибок в программе — это очень дорогостоящее, неприятное и утомительное занятие. Поэтому есть методики, уменьшающие количество ошибок и позволяющие избежать их еще на этапе создания программы. Одна из них — это **проектирование по контракту**. **Design by contract** — это метод проектирования, основанный на идее взаимных обязательств и преимуществ взаимодействующих элементов программы.

- Взаимодействующие элементы программы:
 - «клиент» — это вызывающая функция, объект или модуль;
 - «поставщик» — это вызываемая функция, объект или модуль.
- «Контракт» между ними — это взаимные
 - обязательства — то, что требуется каждой стороне соблюсти при взаимодействии;
 - преимущества — та выгода, которая получается при соблюдении обязательств другой стороной.

Как и в бизнесе, клиент и поставщик действуют в соответствии с определенным контрактом. Архитектор программы должен определить формальные, точные и верифицируемые спецификации интерфейсов для функций и методов.

Содержание контракта

- Предусловия — обязательства клиента перед вызовом функции-поставщика услуги;
- Постусловия — обязательства функции-поставщика, которые обязаны быть выполнены в итоге её работы;
- Инварианты — условия, которые должны выполняться как при вызове функции-поставщика, так и при окончании его работы.

Предусловия, постусловия, инварианты записываются через **формальные утверждения корректности — assertions**:

- Синтаксис:
assert условие формальной корректности, "Сообщение об ошибке!" ;
- Пример:
assert 0 <= hour <= 23, "Hours should be in range of 0..23" ;
- Жёсткое падение ("fail-fast") облегчает проверку выполнения контрактов во время отладки программы;

- Стандартные `assert` работают как в режиме отладки, так и в готовом продукте.

Библиотека **PyContracts** позволяет элегантно ввести в Python элементы проектирования по контракту (в том числе и проверку типов).
Способы описания предусловий:

1. через параметры декоратора:

```
@contract(a='int,>0',
          b='list[N],N>0',
          returns='list[N]')
def my_function(a, b):
    pass
```

2. через аннотации типов

```
@contract
def my_function(a: 'int,>0',
               b: 'list[N],N>0') -> 'list[N]':
    pass
```

3. через документ-строки

```
@contract
def my_function(a, b):
    """ Function description.
        :type a: int,>0
        :type b: list[N], N>0
        :rtype: list[N]
    """
    pass
```

Пример контракта для функции умножения матриц:

```
@contract
def matrix_multiply(a, b):
    """ Multiplies two matrices together.

        :param a: The first matrix. 2D array.
        :type a: array[MxN],M>0,N>0

        :param b: The second matrix. 2D array
                   of compatible dimensions.
        :type b: array[NxP],P>0
```

```

        :rtype: array[MxP]
    """
    return numpy.dot(a, b)

```

Обратите внимание: можно потребовать не только положительного значения ширины или высоты матрицы, но и того, чтобы у матриц *a* и *b* соответствующие размеры были равны друг другу.

Библиотека PyContracts позволяет еще и отключить все проверки, когда ваша программа будет отлажена и отправлена в релиз. Это делается вызовом функции:

```
contracts.disable_all()
```

Или установкой переменной окружения

```
DISABLE_CONTRACTS
```

Плюсы контрактного программирования:

- улучшает дизайн программы;
- повышает надёжность работы программы;
- повышает читаемость кода, поскольку контракт документирует обязательства функций и объектов;
- увеличивает шанс повторного использования кода;
- актуализирует документацию программного продукта.

1.3.3 Модульное тестирование и Test-Driven Development

Декомпозиция программы на функции, классы и модули позволяет осуществлять модульное или компонентное тестирование (unit testing). Юнит-тесты — это способ проверить работу функции или метода отдельно от всей программы, вместе взятой. Рассмотрим юнит тестинг пузырьковой сортировки:

```

def bubble_sort(A):
    """
    Sort on place with Bubblesort algorithm
    :type A: list
    """
    for bypass in range(1, len(A)):
        for k in range(0, len(A) - bypass):
            if A[k] > A[k+1]:
                A[k], A[k+1] = A[k+1], A[k]

def main():

```



```
A = input("Enter some words:").split()
bubble_sort(A)
print("Sorted words:", ' '.join(A))
```

```
main()
```

Поскольку тестирование вручную:

1. займёт время;
2. придётся делать вручную каждый раз;
3. и при этом основная часть программы может ещё не работать.

Нам будет удобнее проверять работу функции отдельно от основного кода и автоматизировать это тестирование. Напишем юнит тест для сортировки:

```
def test_sort():
    A = [4, 2, 5, 1, 3]
    B = [1, 2, 3, 4, 5]
    bubble_sort(A)
    print("#1: " , "Ok" if A == B else "Fail")
```

```
test_sort()
```

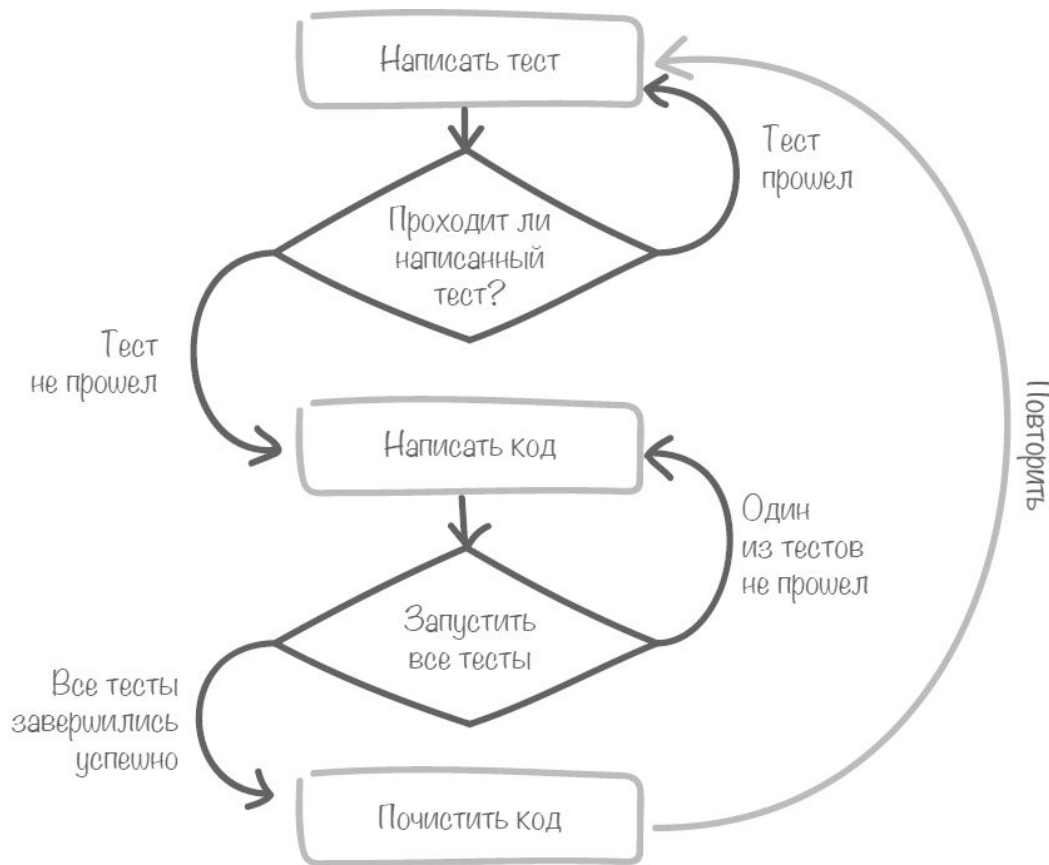
Для большей гарантии стоит добавить несколько различных тестов, которых достаточно для проверки работоспособности функции.

```
def test_sort():
    A = [4, 2, 5, 1, 3]
    B = [1, 2, 3, 4, 5]
    bubble_sort(A)
    print("#1: " , "Ok" if A == B else "Fail")

    A = list(range(40,80)) + list(range(40))
    B = list(range(80))
    bubble_sort(A)
    print("#2: " , "Ok" if A == B else "Fail")

    A = [4, 2, 4, 2, 1]
    B = [1, 2, 2, 4, 4]
    bubble_sort(A)
    print("#3: " , "Ok" if A == B else "Fail")
```

```
test_sort()
```



Опережающее тестирование — способ совмещения роли тестировщика и программиста, при котором сначала пишутся unit-тесты, а потом уже пишется код согласно техническому заданию.

Разработка через тестирование или Test-Driven Development (TDD) — это итеративная методика разработки программ, в которой (опережающее) тестирование ставится во главу угла. И оно управляет процессом дизайна программного продукта. Если существующие тесты проходят нормально, значит, в коде нет известных проблем. Создав тест для выявления недостающего функционала, мы чётко выявляем задачу, которую собираемся решить. И вот такими циклами разработки мы двигаем проект вперёд, создавая новую и новую функциональность программы, которая всегда гарантированно покрыта модульными тестами.

Кроме этого, разработка тестов выявляет **дефекты дизайна приложений**:

- Каковы обязанности тестируемой системы?
- Что и когда она должна делать?
- Какой API удобен для того, чтобы тестируемый код выполнял задуманное?
- Что нужно тестируемой системе для выполнения своих обязательств?
- Что мы имеем на выходе?

- Какие есть побочные эффекты работы?
- Как узнать, что система работает правильно?
- Достаточно ли хорошо определена эта правильность?

Преимущества TDD:

1. Эффективное совмещение ролей (тестирование собственного кода);
2. Рефакторинг без риска испортить код;
3. Реже нужно использовать отладчик;
4. Повышает уверенность в качестве программного кода.

Но работы станет больше: вместо одной функции, придётся писать две (саму функцию и её юнит-тест). Зато будет затрачено меньше времени на поиск ошибок.

[Пример разработки через тестирование](#)

1.3.4 Библиотека `doctest`

Разберём быстрый и удобный способ тестирования программы на примере функции проверки корректности скобочной структуры. Допустим, интерфейс уже проработан. С помощью библиотеки **`doctest`** мы можем сделать примеры из документ-строки действующими (выполняться как юнит-тесты). Добавим несколько примеров корректных и некорректных скобочных последовательностей. И оформим это как если бы мы вызывали это в интерпретаторе:

```
import stack

def is_braces_sequence_correct(seq: str) -> bool:
    """
    Check correctness of braces sequence in statement
    >>> is_braces_sequence_correct("() (())")
    True
    >>> is_braces_sequence_correct("() [()]")
    True
    >>> is_braces_sequence_correct("")
    False
    >>> is_braces_sequence_correct("[()")
    False
    >>> is_braces_sequence_correct("[()]")
    False
    """
```

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Подключили библиотеку doctest и вызвали оттуда функцию testmod:

```
Got nothing
*****
File "D:/Google Drive/Документы/Tex/Python/1 неделя (Хирьянов)/исходные коды/braces.py",
Failed example:
    is_braces_sequence_correct("() [()]" )
Expected:
    True
Got nothing
*****
File "D:/Google Drive/Документы/Tex/Python/1 неделя (Хирьянов)/исходные коды/braces.py",
Failed example:
    is_braces_sequence_correct("") )
Expected:
    False
Got nothing
*****
File "D:/Google Drive/Документы/Tex/Python/1 неделя (Хирьянов)/исходные коды/braces.py",
Failed example:
```

Рис. 1.1: До добавления реализации

В результате для каждого завалившегося теста мы получили отчёт о выполнении этих комментариев. Пока реализации нет, для всех видим "Expected: ... Got nothing" Допишем реализацию:

```
import stack

def is_braces_sequence_correct(seq: str) -> bool:
    """
    Check correctness of braces sequence in statement
    >>> is_braces_sequence_correct("() (())")
    True
    >>> is_braces_sequence_correct("() [()]" )
    True
    >>> is_braces_sequence_correct("") )
    False
    >>> is_braces_sequence_correct("[())" )
    False
    >>> is_braces_sequence_correct("[()]" )
    False
    """
```

```

correspondent = dict(zip("([{", ")]}"))
for brace in seq:
    if brace in "([{":
        stack.push(brace)
        continue
    elif brace in ")]}":
        if stack.is_empty():
            return False
        left = stack.pop()
        if correspondent[left] != brace:
            return False

return stack.is_empty()

if __name__ == "__main__":
    import doctest
    doctest.testmod()

```

При запуске тестов мы увидим только: "Process finished with exit code 0". Тестирование производится, но мы не видим ошибок, ведь всё работает хорошо.

Мы прямо в процессе программирования, еще разрабатывая интерфейс функции и прописывая документ-строку, уже написали тесты простым и естественным способом.

1.3.5 Библиотека unittest

Рассмотрим более сложное тестирование при помощи библиотеки **unittest** на примере функции, вычисляющей числа Фибоначчи.

```

def fib(n: int) -> int:
    """
    Calculates fibonacci number by it's index
    """
    pass

```

Покроем эту функцию модульными тестами в отдельном файле, каждый из которых будет вызовом специального метода.

Для этого создадим тестирующий модуль, который будет содержать тестирующий класс, наследующийся от `unittest.TestCase`. И все методы, начинающиеся со слова `test`, будут тестирующими.

```

import unittest
from fibonacci import fib

```

```
class TestFibonacciNumbers(unittest.TestCase):
    def test_zero(self):
        self.assertEqual(fib(0), 0)
```

Простой юнит тест, проверяющий что число Фибоначчи от 0 это действительно 0 (мы так захотели). `assertEqual` проверяет равенство своих параметров. Модуль `unittest` автоматически запустит этот `test_zero` и оформит результат красиво, если тест будет заваливаться.

Другой вариант использования библиотеки `unittest`: если в вашем модуле достаточно много микротестов(подслучаев), и если у вас произойдет заваливание на одном подслучае, то весь этот маленький тест-кейс просто будет выброшен. А можно было бы протестировать их все, чтобы даже было видно, с каким номером это происходило. Это делается при помощи вызова функции `subTest` с указанием некоторого индекса. Проверим первые 1-5 и 10-ое числа Фибоначчи:

```
...
def test_simple(self):
    for n, fib_n in (1, 1), (2, 1), (3, 2), \
                    (4, 3), (5, 5):
        with self.subTest(i=n):
            self.assertEqual(fib(n), fib_n)

def test_positive(self):
    self.assertEqual(fib(10), 55)
```

Когда производится анализ тестовых случаев, мы должны задуматься, а не могут ли нашу функцию вызвать с какими-то некорректными параметрами, например, для отрицательных чисел.

Давайте в этом случае выбрасывать исключение. Библиотека `unittest` позволяет перехватить это исключение. Добавим таким образом тесты на отрицательные и дробные числа.

```
...
def test_negative(self):
    with self.subTest(i=1):
        self.assertRaises(ArithmeticError, fib, -1)
    with self.subTest(i=1):
        self.assertRaises(ArithmeticError, fib, -10)

def test_fractional(self):
    self.assertRaises(ArithmeticError, fib, 2.5)
```

Отличие `assert`-ов в 1 случае (`test_simple`) и во 2 (`test_negative`) в том, что в `test_simple` мы вызываем функцию самостоятельно, а во втором передаём функцию и список её аргументов.

Если сейчас запустить программу, возникнет целый отчёт, где, что и почему упало:

```
=====
FAIL: test_simple (__main__.TestFibonacciNumbers) (i=3)
=====
Traceback (most recent call last):
  File "D:/Google Drive/Документы/Тех/Python/1 неделя (Хирьянов)/исходные коды/fibonacci_test.py", line 13, in test_simple
    self.assertEqual(fib(n), fib_n)
AssertionError: None != 2

=====
FAIL: test_simple (__main__.TestFibonacciNumbers) (i=4)
=====
Traceback (most recent call last):
  File "D:/Google Drive/Документы/Тех/Python/1 неделя (Хирьянов)/исходные коды/fibonacci_test.py", line 13, in test_simple
    self.assertEqual(fib(n), fib_n)
AssertionError: None != 3

=====
FAIL: test_simple (__main__.TestFibonacciNumbers) (i=5)
=====
Traceback (most recent call last):
  File "D:/Google Drive/Документы/Тех/Python/1 неделя (Хирьянов)/исходные коды/fibonacci_test.py", line 13, in test_simple
    self.assertEqual(fib(n), fib_n)
AssertionError: None != 5

=====
FAIL: test_zero (__main__.TestFibonacciNumbers)
=====
Traceback (most recent call last):
```

В результате он сообщает, что пять тестов прошло и десять ошибок найдено. Анализ этих ошибок позволит найти необходимые поправки.

Напишем нашу функцию правильно:

```
def fib(n: int) -> int:
    """
    Calculates fibonacci number by it's index
    """
    f = [0, 1] + [0] * (n - 1)
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]
    return f[n]
```

Теперь при запуске тестирующего модуля мы поймем ошибки `test_negative` и `test_fractional`. Допишем в начале строчки:

```
def fib(n: int) -> int:
    """
    Calculates fibonacci number by it's index
    """
    if not isinstance(n, int) or n < 0:
        raise ArithmeticError
    f = [0, 1] + [0] * (n - 1)
```

```
for i in range(2, n + 1):  
    f[i] = f[i - 1] + f[i - 2]  
return f[n]
```

После исправления все тесты проходят и мы получаем короткий отчёт:

```
.....  
-----  
Ran 5 tests in 0.000s  
  
OK  
  
Process finished with exit code 0
```

Еще пример покрытия функции юнит-тестами