**Императивная парадигма программирования**

Эта парадигма основана на том, что разработчик (как и тестировщик) пишет для компьютера инструкции, которым тот следует.

К императивной парадигме относятся, например:

* + процедурное;
  + структурное;
  + аспектно-ориентированное;
  + объектно-ориентированное программирование.

## ****Декларативная парадигма программирования****

Эта парадигма отличается тем, что вы описываете проблему и ожидаемый результат, но не создаёте никаких инструкций: отсутствуют переменные, состояние и прочие понятия, которые свойственны императивной парадигме.

К декларативной парадигме относятся **функциональное** и **логическое** программирование.

**Принципы программирования**

Говоря о стиле программирования, стоит вспомнить о **нескольких принципах**, соблюдение которых может сделать ваш код лучше:

1. ***YAGNI****—****Y****ou****A****in’t****G****onna****N****eed It.*
2. ***KISS****—****K****eep****I****t****S****imple,****S****tupid.*
3. ***SOLID****—****S****ingle responsibility principle/****O****pen/closed principle/****L****iskov substitution principle/****I****nterface segregation principle/****D****ependency inversion principle.*
4. ***DRY****—****D****on’t****R****epeat****Y****ourself.*

## YAGNI значит You Ain’t Gonna Need It (вам это не понадобится)

**Суть принципа**: реализуйте только поставленные задачи! Не делайте то, что не нужно.

Избегайте сложных шаблонов, пакетов, фреймворков, которые не будут использоваться или не имеют отношения к работе.

## KISS значит Keep it simple, stupid (делайте вещи проще)

**Суть принципа**: люди (включая пользователей продуктов и услуг), как правило, хотят простых вещей, то есть простых в освоении и использовании. Избегайте чрезмерного проектирования и усложнения!

## DRY значит Don’t repeat yourself (не повторяйте себя)

Иногда можно встретить другой принцип — ***DIE*** (Duplication Is Evil, дублирование — это зло), но смысл у них один.

**Суть принципа**: нужно избегать повторений одного и того же кода. Лучше использовать универсальные свойства и функции.

## SOLID (ещё один принцип)

Это самый длинный акроним в плане расшифровки, под ним скрывается несколько отдельных принципов. Давайте разбираться.

***Single responsibility principle***— принцип единственной обязанности (на каждый класс должна быть возложена одна единственная обязанность). Это как при распределении задач внутри коллектива. Эффективная работа строится тогда, когда каждый выполняет свою определённую задачу.

Например, вам нужно организовать комплексное тестирование сложного продукта. Можно всё взять на себя, а можно грамотно делегировать обязанности и сделать всё быстрее и качественнее.

***Open/closed principle*** — принцип открытости/закрытости. Код, который вы пишете, должен быть масштабируемым, то есть должна быть возможность для его расширения. Но при этом внесение добавлений не должно требовать изменений в коде, который уже написан.

***Liskov substitution principle*** — принцип подстановки [**Барбары Лисков**](https://ru.wikipedia.org/wiki/Лисков,_Барбара). Это принцип объектно-ориентированного программирования, касающийся классов, интерфейсов, типов и подтипов. Суть его в том, что каждый подтип должен дополнять, а не заменять базовый тип.

***Interface segregation principle***— принцип разделения интерфейса. Ни один клиент не должен зависеть от методов, которые он не использует.

**Проще говоря**: не добавляйте дополнительные функции в существующий интерфейс, добавляя новые методы.

***Dependency inversion principle***— [принцип](https://techrocks.ru/2020/04/13/dependency-inversion-principle/)инверсии зависимостей.

В упрощённом варианте применение *SOLID* означает, что мы получаем выгоду от **многоразового, обслуживаемого, масштабируемого**и **легко тестируемого** кода.

**Классы**

**class** **User**: создание класса

**pass**

а = User() – создание экземпляра класса

**User.\_\_dict\_\_** - показывает все атрибуты класса

**Getattr(<вводится название объекта(или класса)>,<”название атрибута”(в виде строки**)>) – позволяет получить значение атрибута. если ввести третий параметр-то он вернет его в том случае если второго атрибута в классе(объекте) нет. Если второй атрибут есть, то он вернет его значение

User.name = “Misha” – добавление или изменение значения атрибута в классе

Setattr(**(<вводится название объекта(или класса)>,<”название атрибута”(в виде строки**)>, <значение атрибута, которое хотим присвоить>) – добавление атрибута к классу

del User.<название атрибута> - удаление атрибута

delattr(User, .<название атрибута>)- удаление атрибута

Класс в процессе изменяется, если в него добавлять или удалять атрибуты. НО экземпляры класса, где бы они не находились, меняются динамически, где бы экземпляр класса не находился, до или после измененний.

**class** **User**:

**pass**

peter = User()

peter.name = "Peter Robertson"

julia = User()

julia.name = "Julia Donaldson"

print(peter.name)

print(julia.name)

Вывод:

Peter Robertson

Julia Donaldson

Здесь .name — это **атрибут** экземпляра нашего класса. Атрибуты позволяют хранить произвольные данные в привязке к конкретному экземпляру

Как создать экземпляр класса User с названием chris?

chris = User() верно

Как создать атрибут phone\_number у объекта peter?

peter.phone\_number = "+79116451238"

## Магический метод \_\_init\_\_

**метод** **\_\_init\_\_**, который заранее определяет атрибуты новых экземпляров. Первым аргументом он обязательно принимает на вход self, а дальше — произвольный набор аргументов, как обычная функция:

**class** **User**:

**def** **\_\_init\_\_**(self, name, email):

self.name = name

self.email = email

Здесь \_\_init\_\_ — это так называемый «магический» метод-конструктор, окруженный двумя нижними подчёркиваниями с обеих сторон. Такие подчёркивания иногда для краткости называются дандерами, от «double underscore». Магические методы обычно неявно вызываются при совершении каких-либо операций, и мы подробнее разберём их дальше в курсе. Пока что для нас неважно, почему он называется именно так — достаточно запомнить, что **конструктор** класса всегда называется именно\_\_init\_\_.

Метод-конструктор задаётся как функция внутри класса (не забудьте про отступ) и первым аргументом всегда принимает self. Именно с помощью этого хитрого финта в него передаётся объект с экземпляром, поэтому его нельзя забывать.

Этот метод вызывается каждый раз, когда мы создаём экземпляр.

После того как мы задали конструктор, при создании объектов в скобки вызова класса можно передавать аргументы, которые он принимает на вход. Чтобы не запутаться, можно явно указать, в какой аргумент что класть:

peter = User(name="Peter Robertson", email="peterrobertson@mail.com")

julia = User(name="Julia Donaldson", email="juliadonaldson@mail.com")

print(peter.name)

print(julia.email)

Вывод:

Peter Robertson

juliadonaldson**@mail**.com

Вроде бы выглядит похоже на то, что мы получили в прошлый раз, но есть ключевая разница: при создании экземпляра класса мы сразу же задаём содержимое его атрибутов. Если так не сделать и не передать в вызов класса нужные аргументы, мы получим ошибку:

>>> chris = User()

Вывод:

**TypeError**: \_\_init\_\_() missing **2** required positional arguments: 'name' **and** 'email'

Мы видим, что конструктор класса способен задавать строгую структуру для создаваемых экземпляров — уже одно это способно здорово помочь с обеспечением единообразия большого количества однотипных объектов. У каждого, как минимум, будет одинаковый набор атрибутов.

## C1.7. Наследование

Помимо того, что классы служат фабрикой по производству однотипных объектов со сходным поведением, они имеют ещё одну важную особенность — наследование.

Как мы уже видели, объекты, созданные при помощи класса, наследуют **атрибуты класса**, которые объявлены прямо в теле, а не добавлены в конкретный независимый экземпляр.

Идея наследования класса состоит в том, что новый класс создаётся не на «пустом месте», а на основе уже существующего. В результате наследования все поля и функции из базового класса неявным образом «наследуются» в производном классе.

При описании производного класса используем шаблон:

**class** **ПроизводныйКласс** (БазовыйКласс):

# Тело класса

**РАБОТА С ФАЙЛАМИ**

*Python* «из коробки» располагает достаточно широким набором инструментов для работы с файлами. Для того чтобы начать работать с файлом, надо его открыть с помощью команды специальной функции open**.**

f = open('path/to/file', 'filemode', encoding='utf8')

path/to/file — путь к файлу может быть относительным или абсолютным. Можно указывать в Unix-стиле (path/to/file) или в Windows-стиле (path\to\file).

1. filemode — режим, в котором файл нужно открывать.

Записывается в виде строки, состоит из следующих букв:

Самый удобный режим: «a+» - это режим, при котором можно и дописывать в конец и читать (.write и .read)

* + - r — открыть на чтение (по умолчанию);
    - w — перезаписать и открыть на запись (если файла нет, то он создастся);(Пример file = open(“<имя файла>“, ”w”, encoding=”utf-8” )). Важно
    - x — создать и открыть на запись (если уже есть — исключение);
    - a — открыть на дозапись (указатель будет поставлен в конец);
    - t — открыть в текстовом виде (по умолчанию);
    - b — открыть в бинарном виде.

1. encoding — указание, в какой кодировке файл записан (utf-8, cp1251 и т. д.). По умолчанию стоит utf-8.

По окончании работы с файлом обязательно выполнять команду file.close(). Если возникнут ошибки-решать их можно с помощью try-exept.

Методы функции open:

**read**

<file>.read() - откроет весь файл. Если указать в аргументе цифру (<file>.read(2)), то он выведет первые два символа в этом файле. Если еще раз вызвать этот метод с тем же параметром, то выведется СЛЕДУЮЩИЕ два символа. Чтобы этого избежать существует следующий метод:

<file>.seek() - метод откатывает наш отсчет вывода read на индекс, который мы передадим в аргумент. Например: вернуться на начало <file>.seek(0)

<file>.readline() – вывод строки, работает как итератор

<file>.readlines() - метод преобразует текст в список, где каждый элемент это строка по порядку

**write**

<file>.write(‘текст’) – перезаписывает открытый файл, добавляя тот текст, который передаем в аргумент. Можно добавлять новый текст по принципу итератора (каждый раз снова вызывая метод).

<file>.writelines(список) — записывает список строк в файл;

## **Менеджер контекста with**

После работы с файлом его нужно закрыть с помощью метода close(). Обработчик данного файла освобождается для операционной системы (если файл был открыт для записи), и другие приложения могут получать к нему доступ. Если не закрыть файл явно, то информация, записываемая в него, может быть утеряна, или же сам файл может повредиться.

Для явного указания места работы с файлом, а также чтобы не забывать закрывать файл после обработки, существует менеджер контекста with.

# В блоке менеджера контекста открытый файл «жив» и с ним можно работать, при выходе из блока файл закрывается.

**with** open("test.txt", 'rb') **as** f:

a = f.read(**10**)

b = f.read(**23**)

f.read(**3**) # Error!

Мы узнали, зачем нужен контекстный менеджер и на практике познакомились с тем, какой он удобный.

1. Контекстный менеджер вызывается с помощью ключевого слова with. Он может возвращать или не возвращать объект для работы. Например, если контекстный менеджер подразумевает работу с каким-либо объектом, то надо добавить в запись: as \*var\*, где var — имя переменной в данном контексте.
2. Мы научились строить собственные контекстные менеджеры на основе классов. Для этого мы пишем класс со специальными методами def \_\_enter\_\_(self, …) и \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb). \_\_enter\_\_ вызывается при входе в контекстный менеджер, \_\_exit\_\_ — при выходе (не забудьте добавить необходимые аргументы).
3. Научились строить контекстные менеджеры на основе генераторов и декоратора contextmanager из библиотеки contextlib.