[2 Объектно-ориентированное программирование 1](#_Toc154481791)

[Классы и объекты 1](#_Toc154481792)

[Инкапсуляция, атрибуты и свойства 8](#_Toc154481793)

[Наследование 12](#_Toc154481794)

[Переопределение функционала базового класса 16](#_Toc154481795)

[Атрибуты классов и статические методы 20](#_Toc154481796)

[Класс object. Строковое представление объекта 24](#_Toc154481797)

[3 Обработка ошибок и исключений 25](#_Toc154481798)

[Конструкция try...except...finally 25](#_Toc154481799)

[except и обработка разных типов исключений 28](#_Toc154481800)

[Генерация исключений и создание своих типов исключений 32](#_Toc154481801)

**2 Объектно-ориентированное программирование**

**Классы и объекты**

Python имеет множество встроенных типов, например, int, str и так далее, которые мы можем использовать в программе. Но также Python позволяет определять собственные типы с помощью **классов**. Класс представляет некоторую сущность. Конкретным воплощением класса является объект.

Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики Человек может выполнять некоторые действия - ходить, бегать, думать и т.д. То есть это представление, которое включает набор характеристик и действий, можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек будет представлять объект этого класса.

Класс определяется с помощью ключевого слова **class**:

|  |  |
| --- | --- |
| 1  2  3 | class название\_класса:      атрибуты\_класса      методы\_класса |

Внутри класса определяются его атрибуты, которые хранят различные характеристики класса, и методы - функции класса.

Создадим простейший класс:

|  |  |
| --- | --- |
| 1  2 | class Person:      pass |

В данном случае определен класс Person, который условно представляет человека. В данном случае в классе не определяется никаких методов или атрибутов. Однако поскольку в нем должно быть что-то определено, то в качестве заменителя функционала класса применяется оператор **pass**. Этот оператор применяется, когда синтаксически необходимо определить некоторый код, однако мы не хотим его, и вместо конкретного кода вставляем оператор pass.

После создания класса можно определить объекты этого класса. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person:      pass    tom = Person()      # определение объекта tom  bob = Person()      # определение объекта bob |

После определения класса Person создаются два объекта класса Person - tom и bob. Для создания объекта применяется специальная функция - **конструктор**, которая называется по имени класса и которая возвращает объект класса. То есть в данном случае вызов Person() представляет вызов конструктора. Каждый класс по умолчанию имеет конструктор без параметров:

|  |  |
| --- | --- |
| 1 | tom = Person()      # Person() - вызов конструктора, который возвращает объект класса Person |

### Методы классов

Методы класса фактически представляют функции, которые определенны внутри класса и которые определяют его поведение. Например, определим класс Person с одним методом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Person:       # определение класса Person       def say\_hello(self):          print("Hello")    tom = Person()  tom.say\_hello()    # Hello |

Здесь определен метод say\_hello(), который условно выполняет приветствие - выводит строку на консоль. При определении методов любого класса следует учитывать, что все они должны принимать в качестве первого параметра ссылку на текущий объект, который согласно условностям называется **self**. Через эту ссылку внутри класса мы можем обратиться к функциональности текущего объекта. Но при самом вызове метода этот параметр не учитывается.

Используя имя объекта, мы можем обратиться к его методам. Для обращения к методам применяется нотация точки - после имени объекта ставится точка и после нее идет вызов метода:

|  |  |
| --- | --- |
| 1 | объект.метод([параметры метода]) |

Например, обращение к методу say\_hello() для вывода приветствия на консоль:

|  |  |
| --- | --- |
| 1 | tom.say\_hello()    # Hello |

В итоге данная программа выведет на консоль строку "Hello".

Если метод должен принимать другие параметры, то они определяются после параметра self, и при вызове подобного метода для них необходимо передать значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Person:       # определение класса Person      def say(self, message):     # метод          print(message)      tom = Person()  tom.say("Hello METANIT.COM")    # Hello METANIT.COM |

Здесь определен метод say(). Он принимает два параметра: self и message. И для второго параметра - message при вызове метода необходимо передать значение.

### self

Через ключевое слово **self** можно обращаться внутри класса к функциональности текущего объекта:

|  |  |
| --- | --- |
| 1  2 | self.атрибут    # обращение к атрибуту  self.метод      # обращение к методу |

Например, определим два метода в классе Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person:        def say(self, message):          print(message)        def say\_hello(self):          self.say("Hello work")  # обращаемся к выше определенному методу say      tom = Person()  tom.say\_hello()     # Hello work |

Здесь в одном методе - say\_hello() вызывается другой метод - say():

|  |  |
| --- | --- |
| 1 | self.say("Hello work") |

Поскольку метод say() принимает кроме self еще параметры (параметр message), то при вызове метода для этого параметра передается значение.

Причем при вызове метода объекта нам обязательно необходимо использовать слово **self**, если мы его не используем:

|  |  |
| --- | --- |
| 1  2 | def say\_hello(self):      say("Hello work")  # ! Ошибка |

То мы столкнемся с ошибкой

### Конструкторы

Для создания объекта класса используется конструктор. Так, выше когда мы создавали объекты класса Person, мы использовали конструктор по умолчанию, который не принимает параметров и который неявно имеют все классы:

|  |  |
| --- | --- |
| 1 | tom = Person() |

Однако мы можем явным образом определить в классах конструктор с помощью специального метода, который называется **\_\_init\_\_()** (по два прочерка с каждой стороны). К примеру, изменим класс Person, добавив в него конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person:      # конструктор      def \_\_init\_\_(self):          print("Создание объекта Person")        def say\_hello(self):          print("Hello")      tom = Person()      # Создание объекта Person  tom.say\_hello()     # Hello |

Итак, здесь в коде класса Person определен конструктор и метод say\_hello(). В качестве первого параметра конструктор, как и методы, также принимает ссылку на текущий объект - self. Обычно конструкторы применяются для определения действий, которые должны производиться при создании объекта.

Теперь при создании объекта:

|  |  |
| --- | --- |
| 1 | tom = Person() |

будет производится вызов конструктора \_\_init\_\_() из класса Person, который выведет на консоль строку "Создание объекта Person".

### Атрибуты объекта

Атрибуты хранят состояние объекта. Для определения и установки атрибутов внутри класса можно применять слово **self**. Например, определим следующий класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person:        def \_\_init\_\_(self, name):          self.name = name    # имя человека          self.age = 1        # возраст человека      tom = Person("Tom")    # обращение к атрибутам  # получение значений  print(tom.name)     # Tom  print(tom.age)      # 1  # изменение значения  tom.age = 37  print(tom.age)      # 37 |

Теперь конструктор класса Person принимает еще один параметр - name. Через этот параметр в конструктор будет передаваться имя создаваемого человека.

Внутри конструктора устанавливаются два атрибута - name и age (условно имя и возраст человека):

|  |  |
| --- | --- |
| 1  2  3 | def \_\_init\_\_(self, name):      self.name = name      self.age = 1 |

Атрибуту self.name присваивается значение переменной name. Атрибут age получает значение 1.

Если мы определили в классе конструктор \_\_init\_\_, мы уже не сможем вызвать конструктор по умолчанию. Теперь нам надо вызывать наш явным образом опреледеленный конструктор \_\_init\_\_, в который необходимо передать значение для параметра name:

|  |  |
| --- | --- |
| 1 | tom = Person("Tom") |

Далее по имени объекта мы можем обращаться к атрибутам объекта - получать и изменять их значения:

|  |  |
| --- | --- |
| 1  2 | print(tom.name)     # получение значения атрибута name  tom.age = 37        # изменение значения атрибута age |

В принципе нам необязательно определять атрибуты внутри класса - Python позволяет сделать это динамически вне кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person:        def \_\_init\_\_(self, name):          self.name = name    # имя человека          self.age = 1        # возраст человека      tom = Person("Tom")    tom.company = "Microsoft"  print(tom.company)  # Microsoft |

Здесь динамически устанавливается атрибут company, который хранит место работы человека. И после установки мы также можем получить его значение. В то же время подобное определение чревато ошибками. Например, если мы попытаемся обратиться к атрибуту до его определения, то программа сгенерирует ошибку:

|  |  |
| --- | --- |
| 1  2 | tom = Person("Tom")  print(tom.company)  # ! Ошибка - AttributeError: Person object has no attribute company |

Для обращения к атрибутам объекта внутри класса в его методах также применяется слово self:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Person:        def \_\_init\_\_(self, name):          self.name = name    # имя человека          self.age = 1        # возраст человека        def display\_info(self):          print(f"Name: {self.name}  Age: {self.age}")      tom = Person("Tom")  tom.display\_info()      # Name: Tom  Age: 1 |

Здесь определяется метод display\_info(), который выводит информацию на консоль. И для обращения в методе к атрибутам объекта применяется слово self: self.name и self.age

### Создание объектов

Выше создавался один объект. Но подобным образом можно создавать и другие объекты класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Person:        def \_\_init\_\_(self, name):          self.name = name    # имя человека          self.age = 1        # возраст человека        def display\_info(self):          print(f"Name: {self.name}  Age: {self.age}")      tom = Person("Tom")  tom.age = 37  tom.display\_info()      # Name: Tom  Age: 37    bob = Person("Bob")  bob.age = 41  bob.display\_info()      # Name: Bob  Age: 41 |

Здесь создаются два объекта класса Person: tom и bob. Они соответствуют определению класса Person, имеют одинаковый набор атрибутов и методов, однако их состояние будет отличаться.

При выполнении программы Python динамически будет определять **self** - он представляет объект, у которого вызывается метод. Например, в строке:

|  |  |
| --- | --- |
| 1 | tom.display\_info()      # Name: Tom  Age: 37 |

Это будет объект tom

А при вызове

|  |  |
| --- | --- |
| 1 | bob.display\_info() |

Это будет объект bob

В итоге мы получим следующий консольный вывод:

Name: Tom Age: 37

Name: Bob Age: 41

## Инкапсуляция, атрибуты и свойства

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Person:      def \_\_init\_\_(self, name):          self.name = name    # устанавливаем имя          self.age = 1        # устанавливаем возраст        def display\_info(self):          print(f"Имя: {self.name}\tВозраст: {self.age}")      tom = Person("Tom")  tom.name = "Человек-паук"       # изменяем атрибут name  tom.age = -129                  # изменяем атрибут age  tom.display\_info()              # Имя: Человек-паук     Возраст: -129 |

Но в данном случае мы можем, к примеру, присвоить возрасту или имени человека некорректное значение, например, указать отрицательный возраст. Подобное поведение нежелательно, поэтому встает вопрос о контроле за доступом к атрибутам объекта.

С данной проблемой тесно связано понятие инкапсуляции. **Инкапсуляция** является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объект из вызывающего кода.

Касательно инкапсуляции непосредственно в языке программирования Python скрыть атрибуты класса можно сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются **свойствами**.

Изменим выше определенный класс, определив в нем свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Person:      def \_\_init\_\_(self, name):          self.\_\_name = name  # устанавливаем имя          self.\_\_age = 1  # устанавливаем возраст        def set\_age(self, age):          if 1 < age < 110:              self.\_\_age = age          else:              print("Недопустимый возраст")        def get\_age(self):          return self.\_\_age        def get\_name(self):          return self.\_\_name        def display\_info(self):          print(f"Имя: {self.\_\_name}\tВозраст: {self.\_\_age}")      tom = Person("Tom")  tom.display\_info()  # Имя: Tom  Возраст: 1  tom.set\_age(-3486)  # Недопустимый возраст  tom.set\_age(25)  tom.display\_info()  # Имя: Tom  Возраст: 25 |

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: self.\_\_name. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса. Например, присвоение значения этому атрибуту ничего не даст:

|  |  |
| --- | --- |
| 1 | tom.\_\_age = 43 |

Потому что в данном случае просто определяется динамически новый атрибут \_\_age, но это он не имеет ничего общего с атрибутом self.\_\_age.

А попытка получить его значение приведет к ошибке выполнения (если ранее не была определена переменная \_\_age):

|  |  |
| --- | --- |
| 1 | print(tom.\_\_age) |

Однако все же нам может потребоваться устанавливать возраст пользователя из вне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута:

|  |  |
| --- | --- |
| 1  2 | def get\_age(self):      return self.\_\_age |

Данный метод еще часто называют геттер или аксессор.

Для изменения возраста определено другое свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def set\_age(self, age):      if 1 < age < 110:          self.\_\_age = age      else:          print("Недопустимый возраст") |

Данный метод еще называют сеттер или мьютейтор (mutator). Здесь мы уже можем решить в зависимости от условий, надо ли переустанавливать возраст.

Необязательно создавать для каждого приватного атрибута подобную пару свойств. Так, в примере выше имя человека мы можем установить только из конструктора. А для получение определен метод get\_name.

### Аннотации свойств

Выше мы рассмотрели, как создавать свойства. Но Python имеет также еще один - более элегантный способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя\_свойства\_геттера.setter**.

Перепишем класс Person с использованием аннотаций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class Person:      def \_\_init\_\_(self, name):          self.\_\_name = name  # устанавливаем имя          self.\_\_age = 1  # устанавливаем возраст        @property      def age(self):          return self.\_\_age        @age.setter      def age(self, age):          if 1 < age < 110:              self.\_\_age = age          else:              print("Недопустимый возраст")        @property      def name(self):          return self.\_\_name        def display\_info(self):          print(f"Имя: {self.\_\_name}\tВозраст: {self.\_\_age}")      tom = Person("Tom")    tom.display\_info()  # Имя: Tom  Возраст: 1  tom.age = -3486  # Недопустимый возраст  print(tom.age)  # 1  tom.age = 36  tom.display\_info()  # Имя: Tom  Возраст: 36 |

Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.

Во-вторых, и сеттер, и геттер называются одинаково - age. И поскольку геттер называется age, то над сеттером устанавливается аннотация @age.setter.

После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение tom.age.

## Наследование

**Наследование** позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является одним из краеугольных камней объектно-ориентированного программирования.

Ключевыми понятиями наследования являются **подкласс** и **суперкласс**. **Подкласс** наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).

Синтаксис для наследования классов выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2 | class подкласс (суперкласс):      методы\_подкласса |

Например, у нас есть класс Person, который представляет человека:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person:        def \_\_init\_\_(self, name):          self.\_\_name = name   # имя человека        @property      def name(self):          return self.\_\_name        def display\_info(self):          print(f"Name: {self.\_\_name} ") |

Предположим, нам необходим класс работника, который работает на некотором предприятии. Мы могли бы создать с нуля новый класс, к примеру, класс Employee:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Employee:        def \_\_init\_\_(self, name):          self.\_\_name = name  # имя работника        @property      def name(self):          return self.\_\_name        def display\_info(self):          print(f"Name: {self.\_\_name} ")        def work(self):          print(f"{self.name} works") |

Однако класс Employee может иметь те же атрибуты и методы, что и класс Person, так как работник - это человек. Так, в выше в классе Employee только добавляется метод works, весь остальной код повторяет функционал класса Person. Но чтобы не дублировать функционал одного класса в другом, в данном случае лучше применить наследование.

Итак, унаследуем класс Employee от класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Person:        def \_\_init\_\_(self, name):          self.\_\_name = name   # имя человека        @property      def name(self):          return self.\_\_name        def display\_info(self):          print(f"Name: {self.\_\_name} ")      class Employee(Person):        def work(self):          print(f"{self.name} works")      tom = Employee("Tom")  print(tom.name)     # Tom  tom.display\_info()  # Name: Tom  tom.work()          # Tom works |

Класс Employee полностью перенимает функционал класса Person, лишь добавляя метод work(). Соответственно при создании объекта Employee мы можем использовать унаследованный от Person конструктор:

|  |  |
| --- | --- |
| 1 | tom = Employee("Tom") |

И также можно обращаться к унаследованным атрибутам/свойствам и методам:

|  |  |
| --- | --- |
| 1  2 | print(tom.name)     # Tom  tom.display\_info()  # Name: Tom |

Однако, стоит обратить внимание, что для Employee НЕ доступны закрытые атрибуты типа \_\_name. Например, мы НЕ можем в методе work обратиться к приватному атрибуту self.\_\_name:

|  |  |
| --- | --- |
| 1  2 | def work(self):      print(f"{self.\_\_name} works")   # ! Ошибка |

### Множественное наследование

Одной из отличительных особенностей языка Python является поддержка множественного наследования, то есть один класс можно унаследовать от нескольких классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #  класс работника  class Employee:      def work(self):          print("Employee works")      #  класс студента  class Student:      def study(self):          print("Student studies")      class WorkingStudent(Employee, Student):        # Наследование от классов Employee и Student      pass      # класс работающего студента  tom = WorkingStudent()  tom.work()      # Employee works  tom.study()     # Student studies |

Здесь определен класс Employee, который представляет сотрудника фирмы, и класс Student, который представляет учащегося студента. Класс WorkingStudent, который представляет работающего студента, не определяет никакого функционала, поэтому в нем определен оператор **pass**. Класс WorkingStudent просто наследует функционал от двух классов Employee и Student. Соответственно у объекта этого класса мы можем вызвать методы обоих классов.

При этом наследуемые классы могут более сложными по функциональности, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class Employee:        def \_\_init\_\_(self, name):          self.\_\_name = name        @property      def name(self):          return self.\_\_name        def work(self):          print(f"{self.name} works")      class Student:        def \_\_init\_\_(self, name):          self.\_\_name = name        @property      def name(self):          return self.\_\_name        def study(self):          print(f"{self.name} studies")      class WorkingStudent(Employee, Student):      pass      tom = WorkingStudent("Tom")  tom.work()      # Tom works  tom.study()     # Tom studies |

## Переопределение функционала базового класса

В прошлой статье класс Employee полностью перенимал функционал класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Person:        def \_\_init\_\_(self, name):          self.\_\_name = name   # имя человека        @property      def name(self):          return self.\_\_name        def display\_info(self):          print(f"Name: {self.\_\_name} ")      class Employee(Person):        def work(self):          print(f"{self.name} works") |

Но что, если мы хотим что-то изменить из этого функционала? Например, добавить работнику через конструктор, новый атрибут, который будет хранить компанию, где он работает или изменить реализацию метода display\_info. Python позволяет переопределить функционал базового класса.

Например, изменим классы следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Person:        def \_\_init\_\_(self, name):          self.\_\_name = name   # имя человека        @property      def name(self):          return self.\_\_name        def display\_info(self):          print(f"Name: {self.\_\_name}")      class Employee(Person):        def \_\_init\_\_(self, name, company):          super().\_\_init\_\_(name)          self.company = company        def display\_info(self):          super().display\_info()          print(f"Company: {self.company}")        def work(self):          print(f"{self.name} works")      tom = Employee("Tom", "Microsoft")  tom.display\_info()  # Name: Tom                      # Company: Microsoft |

Здесь в классе Employee добавляется новый атрибут - self.company, который хранит компания работника. Соответственно метод \_\_init\_\_() принимает три параметра: второй для установки имени и третий для установки компании. Но если в базом классе определен конструктор с помощью метода \_\_init\_\_, и мы хотим в производном классе изменить логику конструктора, то в конструкторе производного класса мы должны вызвать конструктор базового класса. То есть в конструкторе Employee надо вызвать конструктор класса Person.

Для обращения к базовому классу используется выражение **super()**. Так, в конструкторе Employee выполняется вызов:

|  |  |
| --- | --- |
| 1 | super().\_\_init\_\_(name) |

Это выражение будет представлять вызов конструктора класса Person, в который передается имя работника. И это логично. Ведь имя работника устанавливается именно в конструкторе класса Person. В самом конструкторе Employee лишь устанавливаем свойство company.

Кроме того, в классе Employee переопределяется метод display\_info() - в него добавляется вывод компании работника. Причем мы могли определить этот метод следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | def display\_info(self):      print(f"Name: {self.name}")      print(f"Company: {self.company}") |

Но тогда строка вывода имени повторяла бы код из класса Person. Если эта часть кода совпадает с методом из класса Person, то нет смысла повторяться, поэтому опять же с помощью выражения **super()** обращаемся к реализации метода display\_info в классе Person:

|  |  |
| --- | --- |
| 1  2  3 | def display\_info(self):      super().display\_info()      # обращение к методу display\_info в классе Person      print(f"Company: {self.company}") |

Затем мы можем вызвать вызвать конструктор Employee для создания объекта этого класса и вызвать метод display\_info:

|  |  |
| --- | --- |
| 1  2 | tom = Employee("Tom", "Microsoft")  tom.display\_info() |

Консольный вывод программы:

Name: Tom

Company: Microsoft

### Проверка типа объекта

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции **isinstance()** мы можем проверить тип объекта. Эта функция принимает два параметра:

|  |  |
| --- | --- |
| 1 | isinstance(object, type) |

Первый параметр представляет объект, а второй - тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает True. Например, возьмем следующую иерархию классов Person-Employee/Student:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | class Person:        def \_\_init\_\_(self, name):          self.\_\_name = name   # имя человека        @property      def name(self):          return self.\_\_name        def do\_nothing(self):          print(f"{self.name} does nothing")      #  класс работника  class Employee(Person):        def work(self):          print(f"{self.name} works")      #  класс студента  class Student(Person):        def study(self):          print(f"{self.name} studies")      def act(person):      if isinstance(person, Student):          person.study()      elif isinstance(person, Employee):          person.work()      elif isinstance(person, Person):          person.do\_nothing()      tom = Employee("Tom")  bob = Student("Bob")  sam = Person("Sam")    act(tom)    # Tom works  act(bob)    # Bob studies  act(sam)    # Sam does nothing |

Здесь класс Employee определяет метод work(), а класс Student - метод study.

Здесь также определена функция act, которая проверяет с помощью функции isinstance, представляет ли параметр person определнный тип, и зависимости от результатов проверки обращается к определенному методу объекта.

## Атрибуты классов и статические методы

### Атрибуты класса

Кроме атрибутов объектов в классе можно определять атрибуты классов. Подобные атрибуты определяются в виде переменных уровня класса. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person:       type = "Person"       description = "Describes a person"      print(Person.type)          # Person  print(Person.description)   # Describes a person    Person.type = "Class Person"  print(Person.type)          # Class Person |

Здесь в классе Person определено два атрибута: type, который хранит имя класса, и description, который хранит описание класса.

Для обращения к атрибутам класса мы можем использовать имя класса, например: Person.type, и, как и атрибуты объекта, мы можем получать и изменять их значения.

Подобные атрибуты являются общими для всех объектов класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Person:       type = "Person"       def \_\_init\_\_(self, name):           self.name = name      tom = Person("Tom")  bob = Person("Bob")  print(tom.type)     # Person  print(bob.type)     # Person    # изменим атрибут класса  Person.type = "Class Person"  print(tom.type)     # Class Person  print(bob.type)     # Class Person |

Атрибуты класса могут применяться для таких ситуаций, когда нам надо определить некоторые общие данные для всех объектов. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person:      default\_name = "Undefined"        def \_\_init\_\_(self, name):          if name:              self.name = name          else:              self.name = Person.default\_name      tom = Person("Tom")  bob = Person("")  print(tom.name)  # Tom  print(bob.name)  # Undefined |

В данном случае атрибут default\_name хранит имя по умолчанию. И если в конструктор передана пустая строка для имени, то атрибуту name передается значение атрибута класса default\_name. Для обращения к атрибуту класса внутри методов можно применять имя класса

|  |  |
| --- | --- |
| 1 | self.name = Person.default\_name |

#### Атрибут класса

Возможна ситуация, когда атрибут класса и атрибут объекта совпадает по имени. Если в коде для атрибута объекта не задано значение, то для него может применяться значение атрибута класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Person:      name = "Undefined"        def print\_name(self):          print(self.name)      tom = Person()  bob = Person()  tom.print\_name()    # Undefined  bob.print\_name()    # Undefined    bob.name = "Bob"  bob.print\_name()    # Bob  tom.print\_name()    # Undefined |

Здесь метод print\_name использует атрибут объект name, однако нигде в коде этот атрибут не устанавливается. Зато на уровне класса задан атрибут name. Поэтому при первом обращении к методу print\_name, в нем будет использоваться значение атрибута класса:

|  |  |
| --- | --- |
| 1  2  3  4 | tom = Person()  bob = Person()  tom.print\_name()    # Undefined  bob.print\_name()    # Undefined |

Однако далее мы можем поменять установить атрибут объекта:

|  |  |
| --- | --- |
| 1  2  3 | bob.name = "Bob"  bob.print\_name()    # Bob  tom.print\_name()    # Undefined |

Причем второй объект - tom продолжит использовать атрибут класса. И если мы изменим атрибут класса, соответственно значение tom.name тоже изменится:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | tom = Person()  bob = Person()  tom.print\_name()    # Undefined  bob.print\_name()    # Undefined    Person.name = "Some Person"     # меняем значение атрибута класса  bob.name = "Bob"                # устанавливаем атрибут объекта  bob.print\_name()    # Bob  tom.print\_name()    # Some Person |

### Статические методы

Кроме обычных методов класс может определять статические методы. Такие методы предваряются аннотацией **@staticmethod** и относятся в целом к классу. Статические методы обычно определяют поведение, которое не зависит от конкретного объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Person:      \_\_type = "Person"        @staticmethod      def print\_type():          print(Person.\_\_type)      Person.print\_type()     # Person - обращение к статическому методу через имя класса    tom = Person()  tom.print\_type()     # Person - обращение к статическому методу через имя объекта |

В данном случае в классе Person определен атрибут класса \_\_type, который хранит значение, общее для всего класса - название класса. Причем поскольку название атрибута предваряется двумя подчеркиваниями, то данный атрибут будет приватным, что защитит от недопустимого изменения.

Также в классе Person определен статический метод print\_type, который выводит на консоль значение атрибута \_\_type. Действие этого метода не зависит от конкретного объекта и относится в целом ко всему классу - вне зависимости от объекта на консоль будет выводится одно и то же значение атрибута \_\_type. Поэтому такой метод можно сделать статическим.

## Класс object. Строковое представление объекта

Начиная с 3-й версии в языке программирования Python все классы неявно имеют один общий суперкласс - **object** и все классы по умолчанию наследуют его методы.

Одним из наиболее используемых методов класса object является метод **\_\_str\_\_()**. Когда необходимо получить строковое представление объекта или вывести объект в виде строки, то Python как раз вызывает этот метод. И при определении класса хорошей практикой считается переопределение этого метода.

К примеру, возьмем класс Person и выведем его строковое представление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person:      def \_\_init\_\_(self, name, age):          self.name = name  # устанавливаем имя          self.age = age  # устанавливаем возраст        def display\_info(self):          print(f"Name: {self.name}  Age: {self.age}")      tom = Person("Tom", 23)  print(tom) |

При запуске программа выведет что-то наподобие следующего:

<\_\_main\_\_.Person object at 0x10a63dc00>

Это не очень информативная информация об объекте. Мы, конечно, можем выйти из положения, определив в классе Person дополнительный метод, который выводит данные объекта - в примере выше это метод display\_info.

Но есть и другой выход - определим в классе Person метод **\_\_str\_\_()** (по два подчеркивания с каждой стороны):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person:      def \_\_init\_\_(self, name, age):          self.name = name  # устанавливаем имя          self.age = age  # устанавливаем возраст        def display\_info(self):          print(self)          # print(self.\_\_str\_\_())     # или так        def \_\_str\_\_(self):          return f"Name: {self.name}  Age: {self.age}"      tom = Person("Tom", 23)  print(tom)      # Name: Tom  Age: 23  tom.display\_info()  # Name: Tom  Age: 23 |

Метод \_\_str\_\_ должен возвращать строку. И в данном случае мы возвращаем базовую информацию о человеке. Если нам потребуется использовать эту информацию в других методах класса, то мы можем использовать выражение self.\_\_str\_\_()

И теперь консольный вывод будет другим:

Name: Tom Age: 23

Name: Tom Age: 23

# 3 Обработка ошибок и исключений

## Конструкция try...except...finally

При программировании на Python мы можем столкнуться с двумя типами ошибок. Первый тип представляют **синтаксические ошибки** (syntax error). Они появляются в результате нарушения синтаксиса языка программирования при написании исходного кода. При наличии таких ошибок программа не может быть скомпилирована. При работе в какой-либо среде разработки, например, в PyCharm, IDE сама может отслеживать синтаксические ошибки и каким-либо образом их выделять.

Второй тип ошибок представляют **ошибки выполнения** (runtime error). Они появляются в уже скомпилированной программе в процессе ее выполнения. Подобные ошибки еще называются исключениями. Например, в прошлых темах мы рассматривали преобразование строки в число:

|  |  |
| --- | --- |
| 1  2  3 | string = "5"  number = int(string)  print(number) |

Данный скрипт успешно скомпилируется и выполнится, так как строка "5" вполне может быть конвертирована в число. Однако возьмем другой пример:

|  |  |
| --- | --- |
| 1  2  3 | string = "hello"  number = int(string)  print(number) |

При выполнении этого скрипта будет выброшено исключение ValueError, так как строку "hello" нельзя преобразовать в число:

|  |  |
| --- | --- |
| 1 | ValueError: invalid literal for int() with base 10: 'hello' |

С одной стороны, здесь очевидно, что строка не представляет число, но мы можем иметь дело с вводом пользователя, который также может ввести не совсем то, что мы ожидаем:

|  |  |
| --- | --- |
| 1  2  3 | string = input("Введите число: ")  number = int(string)  print(number) |

При возникновении исключения работа программы прерывается, и чтобы избежать подобного поведения и обрабатывать исключения в Python есть конструкция **try..except**.

### try..except

Конструкция **try..except** имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4 | try:      инструкции  except [Тип\_исключения]:      инструкции |

Весь основной код, в котором потенциально может возникнуть исключение, помещается после ключевого слова **try**. Если в этом коде генерируется исключение, то работа кода в блоке try прерывается, и выполнение переходит в блок **except**.

После ключевого слова **except** опционально можно указать, какое исключение будет обрабатываться (например, ValueError или KeyError). После слова except на следующей стоке идут инструкции блока except, выполняемые при возникновении исключения.

Рассмотрим обработку исключения на примере преобразовании строки в число:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except:      print("Преобразование прошло неудачно")  print("Завершение программы") |

Вводим строку:

Введите число: hello

Преобразование прошло неудачно

Завершение программы

Как видно из консольного вывода, при вводе строки вывод числа на консоль не происходит, а выполнение программы переходит к блоку except.

Вводим правильное число:

Введите число: 22

Введенное число: 22

Завершение программы

Теперь все выполняется нормально, исключение не возникает, и соответственно блок except не выполняется.

### Блок finally

При обработке исключений также можно использовать необязательный блок **finally**. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except:      print("Преобразование прошло неудачно")  finally:      print("Блок try завершил выполнение")  print("Завершение программы") |

Как правило, блок finally применяется для освобождения используемых ресурсов, например, для закрытия файлов.

**Вопрос 1**

Каким будет результат следующей программы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def func():      try:          print("Hello")          return      finally:          print("World")    func() |

Варианты ответов

* Hello World
* Hello
* World
* Программа завершится ошибкой
* Ничего из вышеперечисленного

Ответ

На консоль будет выведено "Hello World". Поскольку в блоке try на консоль выводится "Hello". После блока try в любом случае выполняется блок finally, в котором на консоль выводится "World".

## except и обработка разных типов исключений

### Встроенные типы исключений

В примере выше обрабатывались сразу все исключения, которые могут возникнуть в коде. Однако мы можем конкретизировать тип обрабатываемого исключения, указав его после слова except:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except ValueError:      print("Преобразование прошло неудачно")  print("Завершение программы") |

В данном случае блок execpt обрабатывает только исключения типа **ValueError**, которые могут возникнут при неудачном преобразовании строки в число.

В Python есть следующие базовые типы исключений:

* **BaseException**: базовый тип для всех встроенных исключений
* **Exception**: базовый тип, который обычно применяется для создания своих типов исключений
* **ArithmeticError**: базовый тип для исключений, связанных с арифметическими операциями (OverflowError, ZeroDivisionError, FloatingPointError).
* **BufferError**: тип исключения, которое возникает при невозможности выполнить операцию с буффером
* **LookupError**: базовый тип для исключений, которое возникают при обращении в коллекциях по некорректному ключу или индексу (например, IndexError, KeyError)

От этих классов наследуются все конкретные типы исключений. В Python обладает довольно большим списком встроенных исключений. Весь этот список можно посмотреть в [документации](https://docs.python.org/3/library/exceptions.html). Перечислю только некоторые наиболее часто встречающиеся:

* **IndexError**: исключение возникает, если индекс при обращении к элементу коллекции находится вне допустимого диапазона
* **KeyError**: возникает, если в словаре отсутствует ключ, по которому происходит обращение к элементу словаря.
* **OverflowError**: возникает, если результат арифметической операции не может быть представлен текущим числовым типом (обычно типом float).
* **RecursionError**: возникает, если превышена допустимая глубина рекурсии.
* **TypeError**: возникает, если операция или функция применяется к значению недопустимого типа.
* **ValueError**: возникает, если операция или функция получают объект корректного типа с некорректным значением.
* **ZeroDivisionError**: возникает при делении на ноль.
* **NotImplementedError**: тип исключения для указания, что какие-то методы класса не реализованы
* **ModuleNotFoundError**: возникает при при невозможности найти модуль при его импорте директивой import
* **OSError**: тип исключений, которые генерируются при возникновении ошибок системы (например, невозможно найти файл, память диска заполнена и т.д.)

И если ситуация такова, что в программе могут быть сгенерированы различные типы исключений, то мы можем их обработать по отдельности, используя дополнительные выражения except. И при возникновении исключения Python будет искать нужный блок except, который обрабатывает данный тип исключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | try:      number1 = int(input("Введите первое число: "))      number2 = int(input("Введите второе число: "))      print("Результат деления:", number1/number2)  except ValueError:      print("Преобразование прошло неудачно")  except ZeroDivisionError:      print("Попытка деления числа на ноль")  except BaseException:      print("Общее исключение")  print("Завершение программы") |

Если возникнет исключение в результате преобразования строки в число, то оно будет обработано блоком except ValueError. Если же второе число будет равно нулю, то есть будет деление на ноль, тогда возникнет исключение ZeroDivisionError, и оно будет обработано блоком except ZeroDivisionError.

Тип **BaseException** представляет общее исключение, под которое попадают все исключительные ситуации. Поэтому в данном случае любое исключение, которое не представляет тип ValueError или ZeroDivisionError, будет обработано в блоке except BaseException:.

Однако, если в программе возникает исключение типа, для которого нет соответствующего блока except, то программа не сможет найти соответствующий блок except и сгенерирует исключение. Например, в следующем случае:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | try:      number1 = int(input("Введите первое число: "))      number2 = int(input("Введите второе число: "))      print("Результат деления:", number1/number2)  except ZeroDivisionError:      print("Попытка деления числа на ноль")  print("Завершение программы") |

Здесь предусмотрена обработка деления на ноль с помощью блока except ZeroDivisionError. Однако если пользователь вместо числа введет некорвертиуремую в число в строку, то возникнет исключение типа ValueError, для которого нет соответствующего блока except. И поэтому программа аварийно завершит свое выполнение.

Python позволяет в одном блоке except обрабатывать сразу несколько типов исключений. В этом случае все типы исключения передаются в скобках:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try:      number1 = int(input("Введите первое число: "))      number2 = int(input("Введите второе число: "))      print("Результат деления:", number1/number2)  except (ZeroDivisionError, ValueError):    #  обработка двух типов исключений - ZeroDivisionError и ValueError      print("Попытка деления числа на ноль или некорректный ввод")    print("Завершение программы") |

### Получение информации об исключении

С помощью оператора **as** мы можем передать всю информацию об исключении в переменную, которую затем можно использовать в блоке except:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except ValueError as e:      print("Сведения об исключении", e)  print("Завершение программы") |

Пример некорректного ввода:

Введите число: fdsf

Сведения об исключении invalid literal for int() with base 10: 'fdsf'

Завершение программы

## Генерация исключений и создание своих типов исключений

### Генерация исключений и оператор raise

Иногда возникает необходимость вручную сгенерировать то или иное исключение. Для этого применяется оператор **raise**. Например, сгенерируем исключение

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | try:      number1 = int(input("Введите первое число: "))      number2 = int(input("Введите второе число: "))      if number2 == 0:          raise Exception("Второе число не должно быть равно 0")      print("Результат деления двух чисел:", number1/number2)  except ValueError:      print("Введены некорректные данные")  except Exception as e:      print(e)  print("Завершение программы") |

Оператору **raise** передается объект BaseException - в данном случае объект Exception. В конструктор этого типа можно ему передать сообщение, которое затем можно вывести пользователю. В итоге, если number2 будет равно 0, то сработает оператор raise, который сгенерирует исключение. В итоге управление программой перейдет к блоку **except**, который обрабатывает исключения типа Exception:

Введите первое число: 1

Введите второе число: 0

Второе число не должно быть равно 0

Завершение программы

### Создание своих типов исключений

В языке Python мы не ограничены только встроенными типами исключений и можем, применяя наследование, при необходимости создавать свои типы исключений. Например, возьмем следующий класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Person:      def \_\_init\_\_(self, name, age):          self.\_\_name = name  # устанавливаем имя          self.\_\_age = age   # устанавливаем возраст        def display\_info(self):          print(f"Имя: {self.\_\_name}  Возраст: {self.\_\_age}") |

Здесь класс Person в конструкторе получает значения для имени и возраста и присваивает их приватным переменным name и age. Однако при создании объекта Person мы можем передать в конструктор некорректное с точки зрения логики значение - например, отрицательное число. Одним из способов решения данной ситуации представляет генерация исключения при передаче некорректных значений.

Итак, определим следующий код программы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class PersonAgeException(Exception):      def \_\_init\_\_(self, age, minage, maxage):          self.age = age          self.minage = minage          self.maxage = maxage        def \_\_str\_\_(self):          return f"Недопустимое значение: {self.age}. " \                 f"Возраст должен быть в диапазоне от {self.minage} до {self.maxage}"      class Person:      def \_\_init\_\_(self, name, age):          self.\_\_name = name  # устанавливаем имя          minage, maxage = 1, 110          if minage < age < maxage:   # устанавливаем возраст, если передано корректное значение              self.\_\_age = age          else:                       # иначе генерируем исключение              raise PersonAgeException(age, minage, maxage)        def display\_info(self):          print(f"Имя: {self.\_\_name}  Возраст: {self.\_\_age}")    try:      tom = Person("Tom", 37)      tom.display\_info()  # Имя: Tom  Возраст: 37        bob = Person("Bob", -23)      bob.display\_info()  except PersonAgeException as e:      print(e)    # Недопустимое значение: -23. Возраст должен быть в диапазоне от 1 до 110 |

В начале здесь определен класс PersonAgeException, который наследуется от класса Exception. Как правило, собственные классы исключений наследуются от класса Exception. Класс PersonAgeException предназначен для исключений, связанных с возрастом пользователя.

В конструкторе PersonAgeException получаем три значения - собственное некорректное значение, которое послужило причиной исключения, а также минимальное и максимальное значения возраста.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class PersonAgeException(Exception):      def \_\_init\_\_(self, age, minage, maxage):          self.age = age          self.minage = minage          self.maxage = maxage        def \_\_str\_\_(self):          return f"Недопустимое значение: {self.age}. " \                 f"Возраст должен быть в диапазоне от {self.minage} до {self.maxage}" |

В функции \_\_str\_\_ определяем текстовое представление класса - по сути сообщение об ошибке.

В конструкторе класса Persoon проверяем переданное для возраста пользователя значение. И если это значение не соответствует определенному диапазону, то генерируем исключение типа PersonAgeException:

|  |  |
| --- | --- |
| 1 | raise PersonAgeException(age, minage, maxage) |

При применении класса Person нам следует учитывать, что конструктор класса может сгенерировать исключение при передаче некорректного значения. Поэтому создание объектов Person обертывается в конструкцию try..except:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try:      tom = Person("Tom", 37)      tom.display\_info()  # Имя: Tom  Возраст: 37        bob = Person("Bob", -23)  # генерируется исключение типа PersonAgeException      bob.display\_info()  except PersonAgeException as e:      print(e)    # Недопустимое значение: -23. Возраст должен быть в диапазоне от 1 до 110 |

И если при вызове конструктора Person будет сгенерировано исключение типа PersonAgeException, то управление программой перейдет к блоку except, который обрабатывает исключения типа PersonAgeException в виде вывода информации об исключении на консоль.