Основы PYTHON

Часть 2

План

- Функции
- Области видимости переменных
- Модули
- Обработка исключений

Функции

Функции представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы. Формальное определение функции:

```
def имя_функции ([параметры]):
    инструкции
```

Определение функции начинается с выражения def, которое состоит из имени функции, набора скобок с параметрами и двоеточия.
Параметры в скобках необязательны.
А со следующей строки идет **блок инструкций**, которые выполняет функция.
Все инструкции функции имеют **отступы** от начала строки.

Например, определение простейшей функции:

```
def say_hello():
    print("Hello")
```

Для **вызова функции** указывается имя функции, после которого в скобках идет передача значений для всех ее параметров. Например:

```
say_hello()
say_hello()
say_hello()
```

Определим и используем **функцию с параметрами**:

```
def say_hello(name):
    print("Hello,",name)

say_hello("Tom")
say_hello("Bob")
say_hello("Alice")
```

Функция принимает параметр name, и при вызове функции мы можем передать вместо параметра какой-либо значение:

Значения по умолчанию

Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

```
def say_hello(name="Tom"):
    print("Hello,", name)

say_hello()
say_hello("Bob")
```

Здесь параметр name является необязательным. И если мы не передаем при вызове функции для него значение, то применяется значение по умолчанию, то есть строка "Tom".

Именованные параметры

При передаче значений функция сопоставляет их с параметрами в том порядке, в котором они передаются. Например, пусть есть следующая функция:

```
def display_info(name, age):
    print("Name:", name, "\t", "Age:", age)

display_info("Tom", 22)
```

При вызове функции первое значение "Tom" передается первому параметру - параметру name, второе значение - число 22 передается второму параметру - age. И так далее.

Использование **именованных параметров** позволяет переопределить порядок передачи:

```
def display_info(name, age):
    print("Name:", name, "\t", "Age:", age)

display_info(age=22, name="Tom")
```

Именованные параметры предполагают указание имени параметра с присвоением ему значения при вызове функции.

Неопределенное количество параметров

С помощью символа звездочки можно определить неопределенное количество параметров:

```
def sum(*params):
    result = 0
    for n in params:
        result += n
    return result

sumOfNumbers1 = sum(1, 2, 3, 4, 5)      # 15
sumOfNumbers2 = sum(3, 4, 5, 6)         # 18
print(sumOfNumbers1)
print(sumOfNumbers2)
```

В данном случае функция sum принимает один параметр - *params, но звездочка перед названием параметра указывает, что фактически на место этого параметра мы можем передать неопределенное количество значений или набор значений. В самой функции

Возвращение результата

Функция может возвращать результат. Для этого в функции используется оператор return, после которого указывается возвращаемое значение:

```
def exchange(usd_rate, money):
    result = round(money/usd_rate, 2)
    return result

result1 = exchange(60, 30000)
print(result1)
result2 = exchange(56, 30000)
print(result2)
result3 = exchange(65, 30000)
print(result3)
```

Поскольку функция возвращает значение, то мы можем присвоить это значение какой-либо переменной и затем использовать ее: result2 = exchange(56, 30000).

В Python функция может возвращать сразу несколько значений:

```
def create_default_user():
    name = "Tom"
    age = 33
    return name, age


user_name, user_age = create_default_user()
print("Name:", user_name, "\t Age:", user_age)
```

Здесь функция create_default_user возвращает два значения: name и age. При вызове функции эти значения по порядку присваиваются переменным user_name и user_age, и мы их можем использовать.

Функция main

В программе может быть определено множество функций. И чтобы всех их упорядочить, хорошей практикой считается добавление специальной функции main, в которой потом уже вызываются другие функции:

```
def main():
    say_hello("Tom")
    usd_rate = 56
    money = 30000
    result = exchange(usd_rate, money)
    print("К выдаче", result, "долларов")
def say_hello(name):
    print("Hello,", name)
def exchange(usd_rate, money):
    result = round(money/usd_rate, 2)
    return result
# Вызов функции main
main()
```

Область видимости переменных

Область видимости или scope определяет контекст переменной, в рамках которого ее можно использовать.

В Python есть два типа контекста:

- глобальный и
- локальный.

Глобальный контекст

Глобальный контекст подразумевает, что переменная является глобальной, она определена вне любой из функций и доступна любой функции в программе. Например:

```
name = "Tom"

def say_hi():
    print("Hello", name)
def say_bye():
    print("Good bye", name)

say_hi()
say_bye()
```

Здесь переменная name является глобальной и имеет глобальную область видимости. И обе определенные здесь функции могут свободно ее использовать.

локальная переменная

В отличие от глобальных переменных локальная переменная определяется **внутри функции** и **доступна только из этой функции**, то есть имеет локальную область видимости:

```
def say_hi():
    name = "Sam"
    surname = "Johnson"
    print("Hello", name, surname)
def say_bye():
    name = "Tom"
    print("Good bye", name)

say_hi()
say_bye()
```

Есть еще один вариант определения переменной, когда локальная переменная скрывают глобальную с тем же именем:

```
name = "Tom"

def say_hi():
    print("Hello", name)
def say_bye():
    name = "Bob"
    print("Good bye", name)

say_hi()  # Hello Tom
say_bye()  # Good bye Bob
```

Если же мы хотим изменить в локальной функции глобальную переменную, а не определить локальную, то необходимо использовать ключевое слово **global**:

```
def say_bye():
    global name
    name = "Bob"
    print("Good bye", name)
```

В Python, как и во многих других языках программирования, не рекомендуется использовать глобальные переменные. Единственной допустимой практикой является определение небольшого числа глобальных констант, которые не изменяются в процессе работы программы.

```
PI = 3.14


# вычисление площади круга
def get_circle_square(radius):

    print("Площадь круга с радиусом", radius, "равна", PI * radius * radius)

get_circle_square(50)
```

Модули

Модуль в языке Python представляет отдельный файл с кодом, который можно повторно использовать в других программах.

создание модуля

Для создания модуля необходимо создать собственно файл с расширением *.py, который будет представлять модуль.
Название файла будет представлять название модуля. Затем в этом файле надо определить одну или несколько функций.

Пусть основной файл программы будет называться hello.py. И мы хотим подключить к нему внешние модули.

Для этого сначала определим новый модуль:
создадим новый файл, который назовем account.py, в той же папке, где находится hello.py.
Если используется PyCharm или другая IDE, то оба файла просто помещаются в один проект.

Соответственно модуль будет называться account.
И определим в нем следующий код:

```
def calculate_income(rate, money, month):
    if money <= 0:
        return 0

    for i in range(1, month+1):
        money = round(money + money * rate / 100 / 12, 2)
    return money
```

использование модуля

Для использования модуля его надо импортировать с помощью оператора **import**, после которого указывается имя модуля:

import account.

Чтобы обращаться к функциональности модуля, нам нужно получить его пространство имен. По умолчанию оно будет совпадать с именем модуля, то есть в нашем случае также будет называться account.

Получив пространство имен модуля, мы сможем обратиться к его функциям по схеме

пространство_имен.функция:

```
account.calculate_income(rate, money, period)
```

И после этого мы можем запустить главный скрипт hello.py, и он задействует модуль account.py.

```
#! Программа Банковский счет
import account

rate = int(input("Введите процентную ставку: "))
money = int(input("Введите сумму: "))
period = int(input("Введите период ведения счета в месяцах: "))

result = account.calculate_income(rate, money, period)
print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ", rate, "\n",
      "Период: ", period, "\n", "Сумма на счете в конце периода: ", result)
```

Настройка пространства имен

По умолчанию при импорте модуля он доступен через одноименное пространство имен. Однако мы можем переопределить это поведение. Так, ключевое слово as позволяет сопоставить модуль с другим пространством имен. Например:

```
import account as acc
#.............

result = acc.calculate_income(rate, money, period)
```

В данном случае пространство имен будет называться acc.

Импорт в глобальное пространство имен

Другой вариант настройки предполагает импорт функциональности модуля в глобальное пространство имен текущего модуля с помощью ключевого слова **from**:

```
from account import calculate_income

#.............

result = calculate_income(rate, money, period)
```

В данном случае мы импортируем из модуля account в глобальное пространство имен функцию calculate_income. Поэтому мы сможем ее использовать без указания пространства имен модуля как если бы она была определена в этом же файле.

Если бы в модуле account было бы несколько функций, то могли бы их импортировать в глобальное пространство имен одним выражением:

```
from account import *

#.............

result = calculate_income(rate, money, period)
```

Но стоит отметить, что импорт в глобальное пространство имен чреват **коллизиями имен функций**. Например, если у нас том же файле определена функция с тем же именем, то при вызове функции мы можем получить ошибку. Поэтому лучше **избегать** использования импорта в глобальное пространство имен.

Имя модуля

В примере выше модуль hello.py, который является главным, использует модуль account.py.
При запуске модуля hello.py программа выполнит всю необходимую работу. Однако, если мы запустим отдельно модуль account.py сам по себе, то ничего на консоли не увидим. Ведь модуль просто определяет функцию и не выполняет никаких других действий.
Но мы можем сделать так, чтобы модуль account.py мог использоваться как сам по себе, так и подключаться в другие модули.

При выполнении модуля среда определяет его имя и присваивает его глобальной переменной **__name__** (с обеих сторон два подчеркивания).
Если модуль является запускаемым, то его имя равно __main__ (также по два подчеркивания с каждой стороны). Если модуль используется в другом модуле, то в

```
def calculate_income(rate, money,
month):
    if money <= 0:
        return 0

    for i in range(1, month+1):
        money = round(money + money
* rate / 100 / 12, 2)
    return money
```

```
  def main():
    rate = 10
    money = 100000
    period = 12
    result = calculate_income(rate,
money, period)
    print("Параметры счета:\n",
"Сумма: ", money, "\n", "Ставка: ",
rate, "\n",  "Период: ", period, "\n",
"Сумма на счете в конце периода: ",
result)

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

Кроме того, для тестирования функции определена главная функция main. И мы можем сразу запустить файл account.py отдельно от всех и протестировать код. Следует обратить внимание на вызов функции main:

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

Переменная __name__ указывает на имя модуля. Для главного модуля, который непосредственно запускается, эта переменная всегда будет иметь значение __main__ вне зависимости от имени файла.

Поэтому, если мы будем запускать скрипт account.py отдельно, сам по себе, то Python присвоит переменной __name__ значение __main__, далее в выражении if вызовет функцию main из этого же файла.

Однако если мы будем запускать другой скрипт, а этот - account.py - будем подключать в качестве вспомогательного, для account.py переменная __name__ будет иметь значение account. И соответственно метод main в файле account.py не будет работать.

Данный подход с проверкой имени модуля является более рекомендуемым подходом, чем просто вызов метода main.

В файле hello.py также можно сделать проверку на то, является ли модуль главным (хотя в прицнипе это необязательно):

```
#! Программа Банковский счет
import account


def main():
    rate = int(input("Введите процентную ставку: "))
    money = int(input("Введите сумму: "))
    period = int(input("Введите период ведения счета в месяцах: "))

    result = account.calculate_income(rate, money, period)
    print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ", rate, "\n",
          "Период: ", period, "\n", "Сумма на счете в конце периода: ", result)

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

Обработка исключений

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**.

преобразование числа в строку:

```
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:

```
string = "hello"
number = int(string)
print(number)
```

При выполнении этого скрипта будет выброшено исключение ValueError, так как строку "hello" нельзя преобразовать в число. С одной стороны, здесь очевидно, сто строка не представляет число, но мы можем иметь дело с вводом пользователя, который также может ввести не совсем то, что мы ожидаем:

```
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, которая имеет следующее *формальное определение*:

```
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:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except:
    print("Преобразование прошло неудачно")
print("Завершение программы")
```

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

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except ValueError:
    print("Преобразование прошло неудачно")
print("Завершение программы")
```

Если ситуация такова, что в программе могут быть сгенерированы различные типы исключений, то мы можем их обработать по отдельности, используя дополнительные выражения except:

```
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 Exception:
    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 Exception 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 Exception:.

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:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except ValueError:
    print("Не удалось преобразовать число")
finally:
    print("Блок try завершил выполнение")
print("Завершение программы")
```

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

Получение информации об исключении

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

```
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

Иногда возникает необходимость вручную сгенерировать то или иное исключение. Для этого применяется оператор raise.

```
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")
```

При вызове исключения мы можем ему передать сообщение, которое затем можно вывести пользователю:

Введите первое число: 1
Введите второе число: 0
Второе число не должно быть равно 0
Завершение программы