Информационные технологии

Основы программирования на Python 3

Каф. ИКТ РХТУ им. Д.И. Менделеева
Ст. преп. Васецкий А.М.

Москва, 2018

Лекция 6. Пользовательские функции и файлы

- Пользовательские функции
- Функции высших порядков
- Файлы
- Обработка исключений

Пользовательские функции

- **Функция** – это отдельный именованный блок кода. Она может принимать любое количество входных параметров и возвращать любое количество результатов.
- Это универсальное средство структурирования программы. В разных языках программирования они называются *подпрограммами или процедурами*

Действия с функциями

- Функцию можно: определить, вызвать.
- Блок функции начинается с ключевого слова ***def***, после которого следуют имя функции и круглые скобки (()).
- Имена функций подчиняются тем же правилам, что и имена переменных
- Аргументы, которые принимает функция, должны находиться внутри скобок.
- Далее идёт двоеточие (:).
- Само тело функции начинается с новой строки с отступом.
- Функции, вычисляющие какое-либо значение, возвращают его с помощью инструкции ***return.***
- Функции, известные как *генераторы*, для передачи возвращаемого значения могут также использовать инструкцию ***yield*** и сохранять свое состояние так, чтобы работа функции могла быть возобновлена позднее

Примеры функций

- Пустая функция без параметров:

```
def emptyfun():
    pass
```

- Вызов функции:

```
emptyfun() # скобки обязательны!
```

- Возврат значения:

```
def feedback():
    return True
```

- Вызывая функцию, мы можем передавать ей следующие типы аргументов:
  - ✔Обязательные аргументы (Required arguments)
  - ✔Аргументы-ключевые слова (Keyword argument)
  - ✔Аргументы по умолчанию (Default argument)
  - ✔Аргументы произвольной длины (Variable-length arguments)

Функция с аргументами

- Функция с одним входным параметром:

```
def mycolor(color):
    print(color)
```

- Функция с позиционными аргументами:

```
def menu(one, two, three):
    return {"1": one, "2": two, "3": three}
print(menu(5, 6, 7))          # {'1': 5, '2': 6, '3': 7}
```

- Аргументы – ключевые слова:

```
def menu(one="1", two="2", three="3"):
    return {"1": one, "2": two, "3": three}
print(menu(one=5, three=6, two=7))
# {'1': 5, '2': 7, '3': 6}
print(menu(5, 6, 7))          # {'1': 5, '2': 6, '3': 7}
def menu(one, two, three="3"):
   return {"1": one, "2": two, "3": three}
print(menu(6, 7, three=5))    # правильно
# m{'1': 6, '2': 7, '3': 5}
print(menu(three=5 , 6, 7))   # неправильно
```

Параметры функции по умолчанию

```
def menu(one, two, three="3"): # one, two – обязательные
    return {"1": one, "2": two, "3": three}
print(menu(6, 7)) # {'1': 6, '2': 7, '3': '3'}
```

Значение аргументов по умолчанию высчитывается, когда функция определяется, а не выполняется. Распространенной ошибкой является использование изменяемого типа данных вроде списка или словаря в качестве аргумента по умолчанию

```
def addme(arg, result=[]):
    result.append(arg)
    print(result) # список будет пуст только при первом вызове
addme("a")    # ['a']
addme("b")    # ['a', 'b']
```

Правильно:

```
def addme(arg):
    result = []
    result.append(arg)
    print(result)
addme("a")  # ['a']
addme("b")  # ['b']
```

Или проведя проверку:

```
def addme(arg, result = None):
    if result is None:
        result = []
    result.append(arg)
    print(result)
addme("a")  # ['a']
addme("b")  # ['b']
```

Вариативность аргументов

```
def mult(x, y):
    return x * y
print(mult(2, 5))          # 10
print(mult([2, 5], 4))       # [2, 5, 2, 5, 2, 5, 2, 5]
print(mult("текст", 3))    # текстттекстттекст
a = mult([3, 5], 4)
a[0] = 7
print(a)                   # [7, 5, 3, 5, 3, 5, 3, 5]
```

Позиционные аргументы *args

- Если символ * будет использован внутри функции с параметром, тогда произвольное количество позиционных аргументов будет сгруппировано в кортеж.

```
def maketuple(*args): # *args – принятое обозначение для произвольного числа параметров
    print("аргументы:", args)
maketuple(1, 2, 3)  # аргументы: (1, 2, 3)
maketuple([1, 2], "a") # аргументы: ([1, 2], 'a')
maketuple()        # аргументы: ()
```

Обязательные аргументы

- Обязательные аргументы располагаются перед ***args:**

```
def maketuple(ess, *args):
    print("обязательный:", ess) # обязательный:d
    print("арг:", args)      # арг: (1, 'a', 3)
maketuple("d", 1, "a", 3)
```

Аргументы-ключи ** kwargs

- **Получение аргументов – ключевых слов с помощью *(**kwargs)***
- (**) используются, чтобы сгруппировать аргументы – ключевые слова в словарь, где имена аргументов станут ключами, а их значения – соответствующими значениями в словаре.
- Пример:

```
def my_kwargs(**kwargs):
    print("аргументы:", kwargs)
my_kwargs(one="1", two="2", three="3")
# аргументы: {'one': '1', 'two': '2', 'three': '3'}
```

Передача аргумента-имени функции

- Функцию, как и любой другой объект можно передать по ссылке:

```
def fun1():     # первая функция
    return 1
def fun2():     # вторая функция
    return 2
def fun(*args):  # управляющая функция
    sum = 0
    for curfun in args: # перебираем аргументы
        sum +=curfun() # обращаемся к функции
    return sum
j = fun(fun1, fun2, fun2, fun1)
print(j) # 6  (=1 + 2 + 2 + 1)
```

Документирование функций

- **Удобочитаемость имеет значение**
- Можно прикрепить документацию к определению функции, включив строку в начало ее тела. Такая строка называется **строкой документации**:

```
def doc():
    """Это строка документации.
    Она может быть довольно длинной"""
    pass
print(help(doc))
# doc()
#     Это строка документации.
#     Она может быть довольно длинной
#
# None
```

Внутренние функции

- В Python можно определить функцию внутри другой функции.

```
def outfun(a, b):
    def infun(c, d): # складываем аргументы
        return c + d
    return infun(a, b)
print(outfun(1, 2))  # 3
```

Альтернативные определения

- Допускается вкладывать определения функций внутрь инструкций *if*, что позволяет производить выбор между альтернативами:

```
if fltest: # Определяет функцию таким способом
    def fun():
else:
    def fun(): # Или таким способом
fun() # Вызов выбранной версии
```

- Инструкции ***def*** не интерпретируются, пока они не будут достигнуты и выполнены потоком выполнения, а программный код внутри инструкции ***def*** не выполняется, пока функция не будет вызвана позднее.

Области видимости переменных

- В **Python** три базовых области видимости переменных:
  - ✔ Глобальная
  - ✔ Локальная
  - ✔ Нелокальная
- Переменные, объявленные **внутри** тела функции, имеют локальную область видимости, а объявленные **вне** тела функции, имеют глобальную область видимости.
- Доступ к локальным переменным имеют только те функции, внутри которых они были объявлены, а доступ к глобальным переменным можно получить по всей программе в любой функции.
- По умолчанию все имена, присваивание которым производится внутри функций, являются **локальными** для этих функций и существуют только во время их выполнения.

Пример локальных и глобальных переменных

```
t = 20
def glob():
    print(t)  # Печатаем глобальную переменную t
def loc():
    t = 22    # Создаем локальную переменную t
    print(t)  # Печатаем локальную переменную t
glob()    # напечатает 20
loc()     # напечатает 22
```

Изменение глобальных переменных

- Изменить глобальную переменную внутри функции нельзя, если она не задана с помощью ключевого слова ***global.***

```
t = 20
def glob():
    global t  # доступ к глобальной переменной t
    t+=1
    print(t)   # Печатаем глобальную переменную t
glob()    # напечатает 21
```

nonlocal

- В Python 3 было добавлено новое ключевое слово под названием ***nonlocal***. С его помощью возможно добавлять переопределение области во внутреннюю область. дополнительно см. **PEP 3104**.
- **nonlocal** позволяет назначать переменные во **внешней области**, но не в глобальной.

```
def counter():
    num = 0
    def incrementer():
        num += 1# Неправильно!
        return num
    return incrementer
def counter():
    num = 0
    def incrementer():
        nonlocal num # Правильно!
        num += 1
        return num
    return incrementer
```

Функции высших порядков

- При функциональном стиле программирования стандартной практикой является ***динамическая генерация*** функционального объекта в процессе исполнения кода, с его последующим вызовом в том же коде.
- **Замыкание (closure)**
- **Частичное применение (partial application)**
- **Карринг (carrying)**
- **Функтор**

Замыкания (closure)

- **Замыкание** – это функция, которая динамически генерируется другой функцией, и они обе могут изменяться и запоминать значения переменных, которые были созданы вне функции.
- Дэвид Мертц приводит следующее определение замыкания: "***Замыкание*** – это процедура вместе с привязанной к ней совокупностью данных" (в противовес объектам в объектном программировании, как: "данные вместе с привязанным к ним совокупностью процедур").
- Замыкание – это более общий случай декоратора.

Смысл замыкания

- Применение замыкания позволяет
  - ✔устранить жестко кодированные константы;
  - ✔убрать глобальные переменные из кода;
  - ✔увеличить производительность (В Python загрузка переменных в SCOPE (локальную область) сравнительно долгий процесс).
- Замкнутые переменные доступны только для чтения. Чтобы обойти это ограничение, нужно замыкать переменные в изменяемые переменные, например, в список. Сами замкнутые переменные нельзя будет перезаписывать, а вот содержимое контейнера возможно.

Пример с замыканием

- Пример вложенной функции БЕЗ замыкания:

```
def outer(outerinp):
    def inner(innerinput):
        return innerinput + " Внутренняя"
    return inner(outerinp)
print(outer("  Вызов: ")) # Вызов:  Внутренняя
```

- Пример вложенной функции С замыканием:

```
def outer(outerinp):
    def inner2():
        return outerinp + " Внутренняя"
    return inner2
a = outer(" Вызов: ")
print(a) # <function outer.<locals>.inner2 at
0x0000000002A221E0>
print(a()) # Вызов:  Внутренняя
```

Параметризации создания функции

Смысл замыкания состоит в том, что определение функции "замораживает" окружающий её контекст на **момент определения**. Это может делаться различными способами, например, за счёт параметризации создания функции.

```
def multiplier(n):  # multiplier возвращает функцию умножения на n
    def mul(k):
        return n * k
    return mul
# mul3 – функция, умножающая на 3
mul3 = multiplier(3)
# mul4 – функция, умножающая на 4
mul4 = multiplier(4)
print(mul3(3), mul4(5)) # 9 20
```

Другой способ создания замыкания

- использование значения параметра по умолчанию в точке определения функции.
- Никакие последующие присвоения значений параметру по умолчанию не приведут к изменению ранее определённой функции, но сама функция может быть переопределена:

```
n = 3
def mult(k, mul=n):
    return k * mul
n = 2   # игнорируется функцией mult
print(mult(5)) # k=5, mul = 3. выведет '15'
n = 10 # игнорируется функцией mult
print(mult(6)) # k=6, mul = 3 выведет '18'
n = 5
# Переопределение функции
mult = lambda k, mul=n: mul * k
print(mult(4)) # k=4, mul = 5
# mult=mul*k=5*4=20
```

Частичное применение (partial application)

- Это процесс применения функции к части ее аргументов. Т.е. функция, которая принимает функцию с несколькими параметрами и возвращает функцию с меньшим количеством параметров.
- **Частичное применение** преобразует функцию от ***n*** аргументов к ***(x – n)***, а **карринг** создает ***n*** функций с **1** аргументом.

Пример частичного применения

```
from functools import partial
def greet(greeting, separator, emphasis, name):
    print(greeting + separator + name + emphasis)

greetgirls = partial(greet, greeting="Привет",
separator=",", emphasis=".") # частичное
применение
greetgirls(name=" Аня")  # Привет, Аня.
greetgirls(name=" Оля") # Привет, Оля.

greetboys = partial(greet, greeting="Hello",
emphasis=".") # частичное применение
greetboys(name=" Иван", separator="...")
# Hello... Иван.
greetboys(name=" Игорь", separator="..")
# Hello.. Игорь.
```

Карринг (curring)

- **Карринг** (или каррирование) – преобразование функции от многих переменных в функцию, берущую свои аргументы по одному.

```
def greet_curried(greeting):
    def greet(name):
        print(greeting + ", " + name)
    return greet

greet_hello = greet_curried("Hello") # каррирование
greet_hello("Игорь")     # Hello, Игорь
greet_hello("Роман")     # Hello, Роман
# вызов напрямую greet_curried:
greet_curried("Hi")("Сергей")  # Hi, Сергей
```

Функтор

- Функтор – это не функция, а объект класса, в котором определён метод с именем **__call__()**. Для экземпляра такого объекта может применяться вызов, точно так же, как это происходит для функций.

(Для самостоятельного изучения)

Файлы

- Файл – это именованная последовательность байтов.
- Для считывания данных из файла он должен быть предварительно открыт.
- После окончания работы с файлом его следует закрыть.
- Файлы могут быть открыты как текстовые или бинарные.
- Файлы, открытые в бинарном режиме возвращают содержимое как объекты ***bytes*** без какого-либо декодирования.
- В текстовом режиме содержимое файла возвращается как ***str***, байты сначала декодируются, используя платформенно-зависимую или же заданную кодировку.

Работа с файлами. Функция open

my_file = open(filename, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True, opener=None) # открываем файл

my_file – возвращаемый поток (file object)
filename – путь-имя файла (абсолютное или относительное к текущей рабочей директории), либо целое, являющееся дескриптором файла.
mode – режим (см. таблицу ниже)
buffering – необязательное целое число, используемое для установки политики буферизации.

0 – выключено (только в бинарном режиме).

1 – построчная буферизация (только для текстового режима).

>1 – число – указание размера в байтах порции буфера фиксированного размера.

Режимы работы с файлами. mode

"r"	открытие на чтение (является значением по умолчанию). Указатель стоит в начале файла.
"w"	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый. Указатель стоит в начале файла.
"x"	открытие на запись, если файла не существует, иначе исключение.
"a"	открытие на дозапись, информация добавляется в конец файла. Указатель стоит в конце файла. Создает файл ***file***, если такового не существует.
"b"	открытие в двоичном режиме.
"t"	открытие в текстовом режиме (является значением по умолчанию).
"+"	открытие на чтение и запись

Составные режимы *mode*

"rb"	Открывает файл для чтения в двоичном формате. Указатель стоит в начале файла.
"r+"	Открывает файл для чтения и записи. Указатель стоит в начале файла.
"rb+"	Открывает файл для чтения и записи в двоичном формате. Указатель стоит в начале файла.
"wb"	Открывает файл для записи в двоичном формате. Указатель стоит в начале файла. Создает файл с именем ***file***, если такового не существует.
"w+"	Открывает файл для чтения и записи. Указатель стоит в начале файла. Создает файл с именем ***file***, если такового не существует.
"wb+"	Открывает файл для чтения и записи в двоичном формате. Указатель стоит в начале файла. Создает файл с именем ***file***, если такового не существует.
"ab"	Открывает файл для добавления в двоичном формате. Указатель стоит в конце файла. Создает файл с именем ***file***, если такового не существует.
"a+"	Открывает файл для добавления и чтения. Указатель стоит в конце файла. Создает файл с именем ***file***, если такового не существует.
"ab+"	Открывает файл для добавления и чтения в двоичном формате. Указатель стоит в конце файла. Создает файл с именем ***file***, если такового не существует.

Параметры функции open. *encoding*

- *encoding* – название кодирования, используемого для декодирования или кодирования файла. Используется только в текстовом режиме. По умолчанию кодирование платформенно-зависимо, но любой text encoding, поддерживаемый Python, может быть использован. См. модуль codecs для списка поддерживаемых кодировок.
- Если не указана, используется системная кодировка: для определения вызывается ***locale.getpreferredencoding(False).***
- При работе с двоичными файлами указывать кодировку не требуется.

Параметры функции open. *errors*

errors – необязательный параметр, определяющий обработку ошибок кодирования/декодирования. Следует использовать только для текстовых файлов.

Стандартные имена:

- ***"strict"*** – выбрасывает исключение ValueError, если есть ошибка кодирования. По умолчанию значение ***None*** действует аналогично.
- ***"ignore"*** – игнорирует ошибки.

ВНИМАНИЕ! Это может привести к потере данных!!!

- ***"replace"*** **–** вызывает вставку маркера замены (например "?"), который будет вставлен, где есть бесформенные (неструктурированные) данные.
- ***"surrogateescape"*** **–** будет представлять любые некорректные байты как кодовые точки в Unicode Private Use Area, находящиеся в диапазоне кодов от U+DC80 до U+DCFF. Эти частные точки кода будут затем переведены назад в те же байты, если обработчик ошибок ***"surrogateescape"*** используется, при записи данных. Это полезно при обработки файлов в неизвестной кодировке.
- ***"xmlcharrefreplace"*** – поддерживается только при записи в файл. Неподдерживаемые кодировкой символы заменяются ссылкой соответствующего XML-символа вида ***&#nnn***;.
- ***"backslashreplace"*** **–** заменяет бесформенные данные *backslashed escape* последовательностями Python.
- ***"namereplace"*** **–** (также поддерживается только при записи) заменяет неподдерживаемые символы на ***\N{...}*** escape-последовательности.

Параметры функции open. *newline*

- контролирует, как работает режим universal newlines (это применяется только в текстовом режиме).
- Допустимо: ***None, "", "\n", "\r" и "\r\n"***
- При чтении ввода из потока, если ***newline = None***, режим ***universal newlines*** включен. Строки во вводе могут заканчиваться ***"\n", "\r" или "\r\n"***, и это переводится в ***"\n"*** перед возвратом к вызывающему.
  - ✔Если это есть ***""***, режим ***universal newlines*** включен, но концы строк возвращаются к вызывающему непереведенными.
  - ✔Если у него любое другое допустимое значение ***("\n", "\r" и "\r\n")***, вводимые строки прерываются только заданной строкой, и концы строк возвращаются к вызывающему непереведенными.
- При записи вывода в поток, если ***newline=None***, любые записываемые символы ***"\n"*** переводятся в системный по умолчанию разделитель строк, *os.linesep*.
  - ✔Если ***newline =""*** или ***"\n"***, перевод не происходит.
  - ✔Если ***newline*** есть любое другое допустимое значение, значение ***("\n", "\r" и "\r\n")***, то любые записываемые символы ***"\n"*** переводятся в заданную строку.

Параметры функции open. *closefd*

- ***closefd*** – флаг необходимости закрытия файлового дескриптора. Используется только, если в ***filename*** указан дескриптор, иначе выбрасывается исключение. Если ***closefd=False***, то дескриптор будет оставлен открытым даже после закрытия файла.
- Если имя файла задано, то ***closefd=True*** (по умолчанию), иначе возникает ошибка.

Параметры функции open. *opener*

opener – Пользовательский объект, поддерживающий вызов, который следует использовать для открытия файла. Данный объект, получая на входе ***filename*** и ***flags***, должен возвращать открытый дескриптор файла (возврат ***os.open*** и ***None*** при этом функционально идентичны).

Атрибуты файлового объекта

- *[...'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']*

```
f = open("some.txt", "w")
print("Имя файла: ", f.name)
print("Файл закрыт: ", f.closed)
print("В каком режиме файл открыт: ", f.mode)
print("Кодировка: ", f.encoding)
print("Дескриптор: ", f.fileno())
print("Интерактивный поток: ", f.isatty())
print("Текущая позиция: ", f.tell())
f.close()
```

```
Имя файла:  some.txt
Файл закрыт:  False
В каком режиме файл открыт:  w
Кодировка:  cp1251
Дескриптор:  3
Интерактивный поток:  False
Текущая позиция:  0
```

Использование менеджера контекста *with*

- В Python имеются менеджеры контекста для очистки объектов, таких как открытые файлы.
- Используется конструкция вида:
  ***with** <выражение> **as переменная**:*
  with open("f.txt", "wt") as f:
  ***   f.write("my text")***
- После того как блок кода, расположенный под менеджером контекста завершится, файл будет автоматически закрыт.

Запись в файл. Метод write

- Метод ***write()*** записывает строку в открытый файл. Строки могут содержать двоичные данные, а не только текст.
- Метод ***write()*** не добавляет символ переноса строки ***("\n")*** в конец файла.

```
my_file = open("some.txt", "w")
my_file.write("Hello, World!")
my_file.close()
```

Запись в файл. Метод writelines

fout.writelines(aList) – Записывает в файл указанную последовательность строк.
aList – Последовательность, которой может являться любой объект, поддерживающий итерирование и производящий строки.
Пример:

```
f = open("some.txt", "w")
a=["0","some", "1"]
f.writelines(a)
f.close()
```

файл **some.txt: 0some1**

f.flush – Инициирует сброс данных из буфера в файл.

Чтение из файла. Метод *read*

file.read(size) – Считывает и возвращает указанное количество данных из файла.
size – максимальное количество данных, которое требуется считать. Если параметр не задан, либо число отрицательное, содержимое файла будет считано полностью.
После достижения конца файла, метод возвращает пустую строку.
В файле ***file.txt*** строка: "The only line in file.\n"

```
with open("file.txt") as f:
    f.read() # "The only line in file.\n"
    f.read() # ""
    f.close()

with open("file.txt") as f:
    f.read(5) # 'The o'
    f.read(3) # 'nly'
    f.close()
```

Чтение из файла построчно

- ***file.readline()*** – считывает одну строку.
- если метод возвращает пустую строку, значит достигнут конец файла; если строка содержит лишь символ \n, значит это просто очередная строка.

```
with open("file.txt") as f:
    for line in f:
    print(line)
```

- ***file.readlines()*** – считывает из файла все строки в список и возвращает его.

Запишем в файл ***file.txt*** строки:

1 line.\n
2 line.\n

И считаем их кодом:

```
with open("file.txt") as f:
    my_lines = list(f)
print(my_lines)          # ['1 line.\\n\n', '2 line.\\n']
```

Позиционирование

- ***seek(offset[, from_what])*** – Перемещает указатель текущей позиции в файле к указанному месту.
- ***offset***: Смещение в байтах, относительно позиции, определяемой аргументом ***from_what***
- ***from_what=0*** : начало смещения.
  - ✔**0** – от начала файла;
  - ✔**1** – от текущей позиции;
  - ✔**2** – от конца файла.

Если файл открыт в режиме добавления данных (***a*** или ***a+***) любые изменения, сделанные функцией ***seek()*** будут отменены при последующей записи.

```
with open("myfile.txt", 'r+') as f:
    f.write("0123456789abcdef")
    f.seek(5) # Перемещаемся к 6-му байту от начала файла.
    f.read(1) # '5'
    f.seek(-3, 2) # Перемещаемся к третьему байту от конца файла.
    f.read(1) # 'd'
```

Позиционирование. *tell*

- ***tell()*** – Возвращает целочисленное значение – текущую позицию указателя в файле относительно его начала.
- Текущая позиция указателя – позиция (количество байт), с которой будет осуществляться следующее чтение/запись.

```
with open("file.txt", "r+") as f:
    f.write("0123456789")
    f.tell() # 10
    f.write('abcdef')
    f.tell() # 16
```

Запись структурированных данных

- Существует ряд форматов, которые можно различить по следующим особенностям.
- Данные с разделителями – символы табуляции ('\t'), запятой (',') или ('|'). Такой формат носит название Delimiter-Separated Values (DSV). Его частный случай Comma-Separated Values (CSV)
- Символы "<" и ">", окружающие *теги*. Примерами могут служить форматы **XML** и **HTML**.
- Знаки препинания. Примером является ***JavaScript Object Notation (JSON)***.
- Выделение пробелами.
- Прочие файлы, например конфигурационные.
- Каждый из этих форматов структурированных файлов может быть считан и записан с помощью как минимум одного модуля Python.

Пример записи CSV

```
import csv
a = [
    ["Аня", "Иванова"],
    ["Таня", "Смирнова"],
    ["Рита", "Кузнецова"],
    ["Стас", "Ежов"],
    ["Оля", "Бялко"],]
with open("out.csv", "wt", encoding="utf-8") as fout: # менеджер контекста
    csvout = csv.writer(fout)
    csvout.writerows(a)
```

В файле:

```
Аня,Иванова

Таня,Смирнова

Рита,Кузнецова

Стас,Ежов

Оля,Бялко
```

Больше функционала для работы с файлами

- Для доступа к более широкому функционалу в работе с файлами в Python, – удаление файлов, создание папок и т.д. следует подключить библиотеку ***os***.
- Сохранение структур данных в файл называется *сериализацией*. Форматы вроде JSON могут требовать наличия пользовательских преобразователей для сериализации всех типов данных программы. Python предоставляет модуль ***pickle***, позволяющий сохранить и восстановить любой объект в специальном бинарном формате.

Исключения

- Во многих языках программирования ошибки отображаются с помощью специальных возвращаемых значений.
- В Python используются *исключения:* т.е. код, который выполняется, когда происходит связанная с ним ошибка.
- Хорошим тоном является использование обработчиков исключений везде, где может быть сгенерировано исключение, чтобы пользователь знал, что происходит.
- Существуют 3 типа ошибок:
  - ✔Синтаксические
  - ✔Ошибки времени выполнения
  - ✔Алгоритмические

Обработка исключений

- Для обработки исключений используется инструкция:

try/except/else/finally

Формат инструкции:

try:
    <Блок, в котором перехватываются исключения>
[except [<Исключение1> [as <объект исключения>]] :
    <инструкции>
[...
except [<ИсключениеN>[as <объект исключения>]]:
    <инструкции>]]
[else:
    <Блок, выполняемый, если исключение не возникло>]
[finally:
    <Блок, выполняемый в любом случае>]

Блок *try*

- Если в ходе исполнения блока try всё пройдёт благополучно и исключений не будет, то ни один из обработчиков не задействуется.
- Если в ходе исполнения инструкций в этом блоке будет выброшено исключение, то начнётся поиск подходящего для него обработчика.
- Поиск исключения ведётся по блокам ***except*** поочерёдно, поэтому более распространённые (базовые) типы исключений лучше ставить после более редких.

Блок *except*

- Блоков ***except*** может быть несколько.
- В каждом блоке можно определить свой механизм обработки для одного или более видов исключений
- Все блоки ***except*** должны иметь тело, в котором реализуется обработка исключения.
- После достижения конца блока исполнение продолжается с места, следующего за всей инструкцией.
- Если существуют вложенные друг в друга обработчики одного и того же исключения, и исключение происходит в теле самого внутреннего блока *try*, то и обработано оно будет только самым внутренним обработчиком.

Блок *except*

- Если блок *except* используется без следующего за ним выражения, то он должен быть последним, потому что в нём производится обработка любых типов.
- Если за блоком следует выражение, то оно будет выполнено. Если полученный в результате выполнения объект «совместим» с исключением, которые собираемся обработать, то будут выполнены инструкции, находящиеся в данном блоке. Объект считается *совместимым с поднятым исключением*, если является классом (непосредственным, либо базовым) исключения, либо кортежем, содержащим элемент, совместимый с исключением.
- Если в ходе поиска обработчик найден не будет, то поиск будет продолжен среди обработчиков окружающего кода по стеку.

Блок *except*

- Если в ходе выполнения выражения в блоке ***except*** случится исключение, то поиск обработчика прерывается. При этом начинается поиск нового обработчика в окружающем коде и по стеку вызова. При этом считается, что вся инструкция ***try except*** породила исключение.
- Если найден подходящий блок ***except***, то исключению назначается имя указанное после ключевого слова ***as***. После чего выполняется код в теле блока.
- Применение пустых предложений ***except*** влечет за собой некоторые проблемы проектирования. Несмотря на удобство, они могут перехватывать нежелательные системные исключения, не связанные с работой вашего программного кода, и по случайности прерывать распространение исключений, предназначенных для других обработчиков
- Предложение ***except Exception*** имеет практически тот же эффект, что и пустое предложение ***except***, но оно не перехватывает исключения, имеющие отношение к завершению программы.

Пример обработки исключения

Простейший пример обработки:

```
s_list = [1, 2, 3]
pos = 5
try:
    s_list[pos]
except:
    print("Введите число между 0 и ", len(s_list)-1, " Введено", pos)
```

Блок *else*

- Инструкции в этом необязательном блоке выполняются по завершению блока ***try*** без исключений, а также без ***return, continue u break***

```
def ex():
    try:
        return "Норма"
    except:
        return "Ошибка"
    else:
        return "else"
print(ex())  # 'Норма'
```

```
def exr():
    try:
        pass
    except:
        return "Ошибка"
    else:
        return "else"
print(exr())  # 'else'
```

Блок *finally*

- Инструкции из этого блока будут выполнены после выполнения всех прочих блоков, в том числе если исключение не было обработано (в этом случае оно будет выброшено повторно в конце блока ***finally*** автоматически) и если в блоке ***try*** присутствуют ***return*** или ***break***. **При этом информация об исключении недоступна**.

Типы исключений

- Все встроенные исключения, не являющиеся фатальными (не требующие прерывания работы интерпретатора), наследуются от типа **BaseException**.

- Подробнее см. документацию
- https://docs.python.org/3/library/exceptions.html?highlight=exception#BaseException

Список исключений

- **SystemExit** – исключение, порождаемое функцией sys.exit при выходе из программы.
- **KeyboardInterrupt** – порождается при прерывании программы пользователем (обычно сочетанием клавиш Ctrl+C).
- **GeneratorExit** – порождается при вызове метода close объекта generator.
- **Exception** – здесь заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
  - ✔**StopIteration** – порождается встроенной функцией ***next***, если в итераторе больше нет элементов.
  - ✔**ArithmeticError** – арифметическая ошибка.
    - **FloatingPointError** – порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
    - **OverflowError** – возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как Python поддерживает длинные числа), но может возникать в некоторых других случаях.
    - **ZeroDivisionError** – деление на ноль.

Список исключений

- ✔**AssertionError** – выражение в функции ***assert*** ложно.
- ✔**AttributeError** – объект не имеет данного атрибута (значения или метода).
- ✔**BufferError** – операция, связанная с буфером, не может быть выполнена.
- ✔**EOFError** – функция наткнулась на конец файла и не смогла прочитать то, что хотела.
- ✔**ImportError** – не удалось импортирование модуля или его атрибута.
- ✔**LookupError** – некорректный индекс или ключ.
  - **IndexError** – индекс не входит в диапазон элементов.
  - **KeyError** – несуществующий ключ (в словаре, множестве или другом объекте).
- ✔**MemoryError** – недостаточно памяти.
- ✔**NameError** – не найдено переменной с таким именем.
  - **UnboundLocalError** – сделана ссылка на локальную переменную в функции, но переменная не определена ранее.

Список исключений

✔**OSError** – ошибка, связанная с системой.
- **BlockingIOError**
- **ChildProcessError** – неудача при операции с дочерним процессом.
- **ConnectionError** – базовый класс для исключений, связанных с подключениями.
  - **BrokenPipeError**
  - **ConnectionAbortedError**
  - **ConnectionRefusedError**
  - **ConnectionResetError**
- **FileExistsError** – попытка создания файла или директории, которая уже существует.
- **FileNotFoundError** – файл или директория не существует.
- **InterruptedError** – системный вызов прерван входящим сигналом.
- **IsADirectoryError** – ожидался файл, но это директория.
- **NotADirectoryError** – ожидалась директория, но это файл.
- **PermissionError** – не хватает прав доступа.
- **ProcessLookupError** – указанного процесса не существует.
- **TimeoutError** – закончилось время ожидания.

Список исключений

- ✔**ReferenceError** – попытка доступа к атрибуту со слабой ссылкой.
- ✔**RuntimeError** – возникает, когда исключение не попадает ни под одну из других категорий.
- ✔**NotImplementedError** – возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
- ✔**SyntaxError** – синтаксическая ошибка.
  - •**IndentationError** – неправильные отступы.
    - –**TabError** – смешивание в отступах табуляции и пробелов.
- ✔**SystemError** – внутренняя ошибка.
- ✔**TypeError** – операция применена к объекту несоответствующего типа.
- ✔**ValueError** – функция получает аргумент правильного типа, но некорректного значения.
- ✔**UnicodeError** – ошибка, связанная с кодированием / раскодированием unicode в строках.
  - •**UnicodeEncodeError** – исключение, связанное с кодированием unicode.
  - •**UnicodeDecodeError** – исключение, связанное с декодированием unicode.
  - •**UnicodeTranslateError** – исключение, связанное с переводом unicode.
- ✔**Warning** – предупреждение.

Получение информации об исключении

- Перед исполнением блока ***except***, данные об исключении сохраняются в модуле ***sys*** и могут быть получены при помощи ***sys.exc_info().***
- возвращает кортеж из трех значений, которые дают информацию об исключениях, обрабатывающихся в данный момент.
- Можно использовать ***sys.last_type, sys.last_value, sys.last_traceback***

Спасибо за внимание