Курс по Python

Подпрограмма

Подпрограмма - средство языка программирования, позволяющее упаковывать и параметризовать функциональность.

Как правило, подпрограмма - это именованный фрагмент программного кода, к которому можно обратиться из другого места программы, однако она может и не иметь имени (называясь в таком случае анонимной).

Определение подпрограммы

Подпрограмма должна быть объявлена и в общем случае содержать:

- имя;
- список имен и типов передаваемых параметров (необязательно);
- тип возвращаемого значения (необязательно).

Если подпрограмма возвращает значение вызывающему коду (одно или несколько), она называется функцией, иначе - процедурой.

Вызов подпрограммы

Для того, чтобы использовать ранее определенную подпрограмму, необходимо в требуемом месте кода произвести ее *вызов*, указав:

- указать имя подпрограммы;
- передать требуемые аргументы (значения параметров).

Код, вызвавший подпрограмму, передает ей управление и ожидает завершения выполнения.

Подпрограмма также может вызывать сама себя, т.е. выполняться рекурсивно.

В настоящее время наиболее часто встречаются следующие способы передачи аргументов:

1. По значению

   Для переменной, переданной по значению создается локальная копия и любые изменения, которые происходят в теле подпрограммы с переданной переменной, на самом деле, происходят с локальной копией и никак не сказываются на самой переменной.

2. По ссылке

   Изменения, которые происходят в теле подпрограммы с переменной, переданной по ссылке, происходят с самой переданной переменной.

Преимущества и недостатки

Главное назначение подпрограмм сегодня - **структуризация программы с целью удобства ее понимания и сопровождения**.

Преимущества использования подпрограмм:

- декомпозиция сложной задачи на несколько более простых подзадач: это один из двух главных инструментов структурного программирование (второй - структуры данных);
- уменьшение дублирования кода и возможность повторного использования кода в нескольких программах - следование принципу DRY «не повторяйся» (*англ.* **D**on't **R**epeat **Y**ourself);
- распределение большой задачи между несколькими разработчиками или стадиями проекта;
- сокрытие деталей реализации от пользователей подпрограммы;
- улучшение отслеживания выполнения кода (большинство языков программирования предоставляет стек вызовов подпрограмм).

Недостатком использования подпрограмм можно считать накладные расходы на вызов подпрограммы, однако современные трансляторы стремятся оптимизировать данный процесс.

Функции в Python

В Python нет формального разделения подпрограмм на функции и процедуры (как, например, в Паскале или Си), и процедурой можно считать функцию, возвращающую пустое значение - в остальном используется единственный термин - функция.

def

Для объявления функции в Python используется ключевое слово `def`:

```
def function_name([parameters]):  # `parameters`: параметры функции (через запятую)
    suite                         # Тело функции
```

На имя функции в Python накладываются такие же ограничения, как и на прочие идентификаторы.

Функции в Python

Примечание

PEP8.

Соглашение рекомендует использовать:

- змеиный_регистр (*англ.* snake_case) для наименования функций: `my_function`;
- пустые строки для отделения функций, а большие блоки кода помещать внутрь функций;
- строки документации.

Функции в Python

В Python все данные - объекты, при вызове в функцию передается ссылка на этот объект. При этом, **мутирующие объекты передаются по ссылке, немутирующие - по значению**.

return

Функция в Python может возвращать результат своего выполнения, используя оператор `return` (например, `return 5`). В случае, если он не был указан или указан пустой оператор `return`, возвращается специальное значение `None`.

Примечание

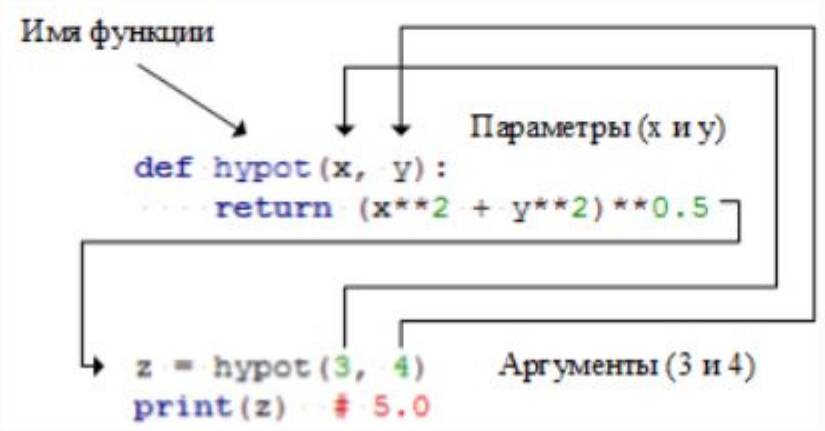
При встрече оператора `return` в коде Python немедленно завершает выполнение функции, аналогично `break` для циклических конструкций.

Функции в Python

```
# Функция для вычисления гипотенузы
# Имя: hypot, 2 параметра: x, y
# return возвращает результат работы функции вызвавшему
def hypot(x, y):
    return (x**2 + y**2)**0.5

z = hypot(3, 4)  # Передача в функцию 2-х аргументов: 3 и 4
print(z)  # 5.0

a = 5
b = 12
print(hypot(a, b))  # 13.0 - результат функции может быть использован сразу
```



Функции в Python

Python, как и многие другие языки, позволяет создавать собственные (пользовательские) функции, среди которых можно выделить четыре типа

1. Глобальные

   Доступны из любой точки программного кода в том же модуле или из других модулей.

2. Локальные (вложенные)

   Объявляются внутри других функций и видны только внутри них: используются для создания вспомогательных функций, которые нигде больше не используются.

3. Анонимные

   Не имеют имени и объявляются в месте использования. В Python и представлены лямбда-выражениями.

4. Методы

   Функции, ассоциированные с каким-либо объектом (например, `list.append()`, где `append()` - метод объекта `list`).

Функции в Python

```
class Car:
    def move(self, x):  # Метод (3)
        self.x += x


def sum_of_cubes(x, y):  # Глобальная функция (1)

    # Локальная функция (2) (ее "видит" только код внутри sum_of_cubes())
    def cube(a):
        return a**3

    return cube(x) + cube(y)  # return возвращает результат выполнения тому,
                              # кто вызвал эту функцию

players = [{"name": "Юрий", "rank": 5},
           {"name": "Сергей", "rank": 3},
           {"name": "Максим", "rank": 4}]
# Анонимная функция (4) (лямбда-выражение)
# В функции sorted() используется для определения порядка сортировки

print(sorted(players, key=lambda player: player["name"]))  # Сортировка по name
# [{'rank': 4, 'name': 'Максим'}, {'rank': 3, 'name': 'Сергей'}, {'rank': 5, 'name': 'Юрий'}]

print(sorted(players, key=lambda player: player["rank"]))  # Сортировка по rank
# [{'rank': 3, 'name': 'Сергей'}, {'rank': 4, 'name': 'Максим'}, {'rank': 5, 'name': 'Юрий'}]
```

Глобальные и локальные функции

Все параметры, указываемые в Python при объявлении и вызове функции делятся на:

- позиционные: указываются простым перечислением:

```
def function_name(a, b, c):  # a, b, c - 3 позиционных параметра
    pass
```

- ключевые: указываются перечислением `ключ=значение`:

```
def function_name(key=value, key2=value2):  # key, key2 - 2 позиционных аргумента
    pass                                    # value, value2 - их значения по умолчанию
```

Глобальные и локальные функции

- позиционные: указываются простым перечислением:

```
def function_name(a, b, c):  # a, b, c - 3 позиционных параметра
    pass
```

- ключевые: указываются перечислением `ключ=значение`:

```
def function_name(key=value, key2=value2):  # key, key2 - 2 позиционных аргумента
    pass                                    # value, value2 - их значения по умолчанию
```

Позиционные и ключевые аргументы могут быть скомбинированы. Синтаксис объявления и вызова функции зависит от типа параметра, однако **позиционные параметры (и соответствующие аргументы) всегда идут перед ключевыми**:

- вызов функции:

```
def example_func(a, b, c=3):  # a, b - позиционные параметры, c - ключевой параметр
    pass

# Вызовы функции
example_func(1, 2, 5)        # можно : аргументы 1, 2, 5 распределяются
                             #         позиционно по параметрам 'a', 'b', 'c'

example_func(1, 2)           # можно : аргументы 1, 2 распределяются позиционно
                             #         по параметрам 'a', 'b'
                             #         в ключевой параметр 'c' аргумент
                             #         не передается, используется значение 3

example_func(a=1, b=2)       # можно : аналогично example_func(1, 2),
                             #         все аргументы передаются по ключу

example_func(b=2, a=1)       # можно : аналогично example_func(a=1, b=2),
                             #         если все позиционные параметры заполнены как
                             #         ключевые аргументы, можно не соблюдать порядок

example_func(c=5, b=2, a=1)  # можно : аналогично example_func(1, 2),
                             #         аргументы передаются по ключу

example_func(1)              # нельзя: для позиционного аргумента 'b'
                             #         не передается аргумент

example_func(b=1)            # нельзя: для позиционного аргумента 'a'
                             #         не передается аргумент
```

Глобальные и локальные функции

Преимущества ключевых параметров:

- нет необходимости отслеживать порядок аргументов;
- у ключевых параметров есть значение по умолчанию, которое можно не передавать.

```
# Функция выдает запрос и
#    - возвращает True в случае положительного ответа
#    - возвращает False в случае отрицательного ответа
#    - возвращает False если не получает ответ за [retries] попыток

def ask_user(prompt, retries=3, hint="Ответьте, ДА или НЕТ?"):
    while True:
        retries -= 1
        ok = input(prompt + " -> ").upper()

        if ok in ("Д", "ДА"):
            return True
        elif ok in ("Н", "НЕТ"):
            return False

        if retries <= 0:
            print("Не смог получить нужный ответ, считаю за отказ.")
            return False
        print(hint)
```

Глобальные и локальные функции

```
# С ключевыми параметрами будут доступны также следующие варианты:
# ask_user("Сохранить файл?", 0)
# ask_user("Сохранить файл?", retries=1)
# ask_user("Сохранить файл?", 2, "Жми Д или Н!!!")
# и др.

if ask_user("Сохранить файл?"):
    print("Сохранил!")
else:
    print("Не сохранил.")

# -------------
# Пример вывода:
#
# Сохранить файл? -> Не знаю
# Ответьте, ДА или НЕТ?
# Сохранить файл? -> Да
# Сохранил!
```

Упаковка и распаковка аргументов

В ряде случаев бывает полезно определить функцию, способную принимать любое число аргументов. Так, например, работает функция `print()`, которая может принимать на печать различное количество объектов и выводить их на экран.

Достичь такого поведения можно, используя механизм **упаковки аргументов**, указав при объявлении параметра в функции один из двух символов:

- `*`: все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж;
- `**`: все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь.

```python
# При упаковке аргументов все переданные позиционные аргументы
# будут собраны в кортеж 'order', а ключевые - в словарь 'info'

def print_order(*order, **info):
    print("Музыкальная библиотека №1\n")

    # Словарь 'infos' должен содержать ключи 'author' и 'birthday'
    for key, value in sorted(info.items()):
        print(key, ":", value)

    # Кортеж 'order' содержит все наименования произведений
    print("Вы выбрали:")
    for item in order:
        print("  -", item)

    print("\nПриходите еще!")

print_order("Славянский марш", "Лебединое озеро", "Спящая красавица",
            "Пиковая дама", "Щелкунчик",
            author="П.И. Чайковский", birthday="07/05/1840")

# -------------
# Пример вывода:
#
# Музыкальная библиотека №1
#
# author : П.И. Чайковский
# birthday : 07/05/1840
# Вы выбрали:
#   - Славянский марш
#   - Лебединое озеро
#   - Спящая красавица
#   - Пиковая дама
#   - Щелкунчик
#
# Приходите еще!
```

Упаковка и распаковка аргументов

Python также предусматривает и обратный механизм - **распаковку аргументов**, используя аналогичные обозначения перед аргументом:

- `*`: кортеж/список распаковывается как отдельные позиционные аргументы и передается в функцию;
- `**`: словарь распаковывается как набор ключевых аргументов и передается в функцию.

Упаковка и распаковка аргументов

```
# Площадь треугольника по формуле Герона
#
# Данный пример функции предназначен для демонстрации распаковки
# Не проектируйте функцию таким образом -
# расчетная функция должна возвращать число, а не строку!
def heron_area_str(a, b, c, units="сантиметры", print_error=True):
    if a + b <= c or a + c <= b or b + c <= a:
        if print_error:
            return "Проверьте введенные стороны треугольника!"
        return

    p = (a + b + c) / 2
    s = (p * (p - a) * (p - b) * (p - c)) ** 0.5
    return "{} {}".format(s, units)

abc = [3, 4, 5]
params = dict(print_error=True, units="см.")

# При распаковке аргументов список 'abc' будет распакован в
# позиционные аргументы, а словарь 'params' - ключевые
print(heron_area_str(*abc, **params))

# -------------
# Пример вывода:
#
# 6.0 см.
```

Область видимости

Область видимости - область программы, где определяются идентификаторы, и транслятор выполняет их поиск. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной, либо быть свободным (не связанным ни с какой из них).

Область видимости

В Python выделяется четыре области видимости:

1. Локальная (*англ.* Local)

   Собственная область внутри инструкции `def`.

2. Нелокальная (*англ.* Enclosed)

   Область в пределах вышестоящей инструкции `def`.

3. Глобальная (*англ.* Global)

   Область за пределами всех инструкций `def` - глобальная для всего модуля.

4. Встроенная (*англ.* Built-in).

   «Системная» область модуля `builtins`: содержит предопределенные идентификаторы, например, функцию `max()` и т.п.

Локальная и нелокальная области видимости являются относительными, глобальная и встроенная - абсолютными.

Область видимости

Основные положения:

- идентификатор может называться локальным, глобальным и т.д., если имеет соответствующую область видимости;
- функции образуют локальную область видимости, а модули – глобальную;
- чем ближе область к концу списка, тем более она открыта (ее содержимое доступно для более закрытых областей видимости; например, глобальные идентификаторы и предопределенные имена могут быть доступны в локальной области видимости функции, но не наоборот).

Область видимости

```
def min_of_cubes(x, y):

    # Идентификаторы 'x' и 'y' являются:
    # - локальными для min_of_cubes()
    # - нелокальными для cube()

    def cube(a):
        return a**3  # 'a' - локальный идентификатор функции cube()

    return min(cube(x), cube(y), cube(c))  # Функция min() находится
                                           # во встроенной области
                                           # видимости и видна везде

# Идентификаторы 'a', 'b' и 'c' имеют глобальную область видимости
a, b, c = 2, 3, 4
print(min_of_cubes(a, b))  # 8
```

Область видимости - примеры

```python
# Увеличиваем переменную 'a' локально
def sum_of_2(a, b):
    a += 2
    return a + b

a, b = 3, 4
print(sum_of_2(a, b))  # 9
print(a, b)            # 3 4
```

Область видимости - примеры

```python
# Добавляем к сумме локальную переменную 'x'
def sum_of_2(a, b):
    x = 5
    return a + b + x

a, b = 3, 4
print(sum_of_2(a, b))  # 12
print(a, b, x)         # NameError: name 'x' is not defined
```

Область видимости - примеры

```
# Пробуем увеличить глобальную переменную 'c'
def sum_of_2(a, b):
    c *= 5  # UnboundLocalError: local variable 'c' referenced before assignment
    return a + b + c

a, b, c = 3, 4, 5
print(sum_of_2(a, b))
```

Область видимости - примеры

```
# Пробуем изменить глобальную переменную 'c'
def sum_of_2(a, b):
    c = 10
    return a + b + c

a, b, c = 3, 4, 5
print(sum_of_2(a, b))  # 17
print(a, b, c)         # 3 4 5
```

Область видимости - примеры

```
# Используем глобальную переменную 'c', не имея соответствующего параметра
def sum_of_2(a, b):
    return a + b + c

a, b, c = 3, 4, 5
print(sum_of_2(a, b))  # 12
```

Область видимости

По умолчанию, идентификаторы из другой области видимости доступны только для чтения, а, при попытке присвоения, функция создает локальный идентификатор.

global

nonlocal

Если *необходимо изменять в функции переменные более закрытой области видимости*, существует 3 способа:

- использовать инструкцию `global`: сообщая, что функция будет изменять один или более глобальных идентификаторов;
- использовать инструкцию `nonlocal`: сообщая, что вложенная функция будет изменять один или более идентификаторов внешних функций;
- передать мутирующий аргумент в качестве параметра функции.

Область видимости

```python
def func():
    # 'value' создается как локальный идентификатор
    value = 100

def func_with_global():
    global value
    # global указывает, что нужно использовать 'value'
    # из глобальной области видимости
    value = 100

value = 0
func()
print(value)  # 0

func_with_global()
print(value)  # 100
```

Область видимости

```
def func():

    def inner_func():
        # 'value' создается как локальный идентификатор inner_func()
        value = 100

    def inner_func_with_nonlocal():
        nonlocal value
        # Благодаря nonlocal используется 'value' из func()
        value = 100

    value = 10
    inner_func()
    print(value)  # 10

    inner_func_with_nonlocal()
    print(value)  # 100

value = 0
func()
print("global value =", value)  # 0
```

Область видимости

```
# Функция принимает список, и добавляет сумму элементов в конец списка
#
# Изменение внутри функции мутирующего объекта приводит к его
# изменению и в вызывающем коде
def sum_list(lst):
    lst.append(sum(lst))

my_list = [1, 2, 3, 4, 5]
sum_list(my_list)
print(my_list)  # [1, 2, 3, 4, 5, 15]
```

Рекурсия

Рекурсия - вызов функции внутри самой себя, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия)

Количество вложенных вызовов функции или процедуры называется *глубиной рекурсии*. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причем без явных повторений частей программы и использования циклов.

Не рекомендуется использовать рекурсию, если такая функция может привести или приводит к большой глубине рекурсии - лучше заменить ее циклической конструкцией. Рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека (*англ.* англ. Stack Overflow) вызовов.

Рекурсия

```python
# Вычисление факториала, используя цикл
def factorial_1(x):
    res = 1
    for i in range(x):
        res *= i + 1
    return res

# Рекурсивный вариант вычисления факториала
def factorial_2(x):
    if x == 1:
        return 1
    else:
        return x * factorial_2(x - 1)

print(factorial_1(5))  # 120
print(factorial_2(5))  # 120
```

Ошибки и исключения

Ошибка (также *баг* от *англ.* Software **Bug**) - неполадка в программе, из-за которой она ведет себя неопределенно, выдавая неожиданный результат.

Основные категории ошибок:

- синтаксические;
- логические;
- ошибки времени выполнения;
- недокументированное поведение.

Синтаксические ошибки

- Причина:

  Несоответствие синтаксису языка программирования. Для компилируемых языков программирования синтаксическая ошибка не позволит выполнить компиляцию.

- Пример:

```
>>> for i in range(10)
  File "<stdin>", line 1
    for i in range(10)
                      ^
SyntaxError: invalid syntax
```

Логические ошибки

- Причина:

  Несоответствие правильной логике работы программы.

- Пример:

```
>>> def avg_of_2(a, b):
...     return a + b / 2
...
>>> avg_of_2(4, 8)  # Вернет 8 вместо 6
```

Ошибки времени выполнения

- Причина:

  Любая неполадка, возникающая во время работы программы, например: целочисленное деление на ноль, ошибка при чтении файла, исчерпание доступной памяти и др.

- Пример:

```
>>> a = 5
>>> b = 0
>>> a / b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

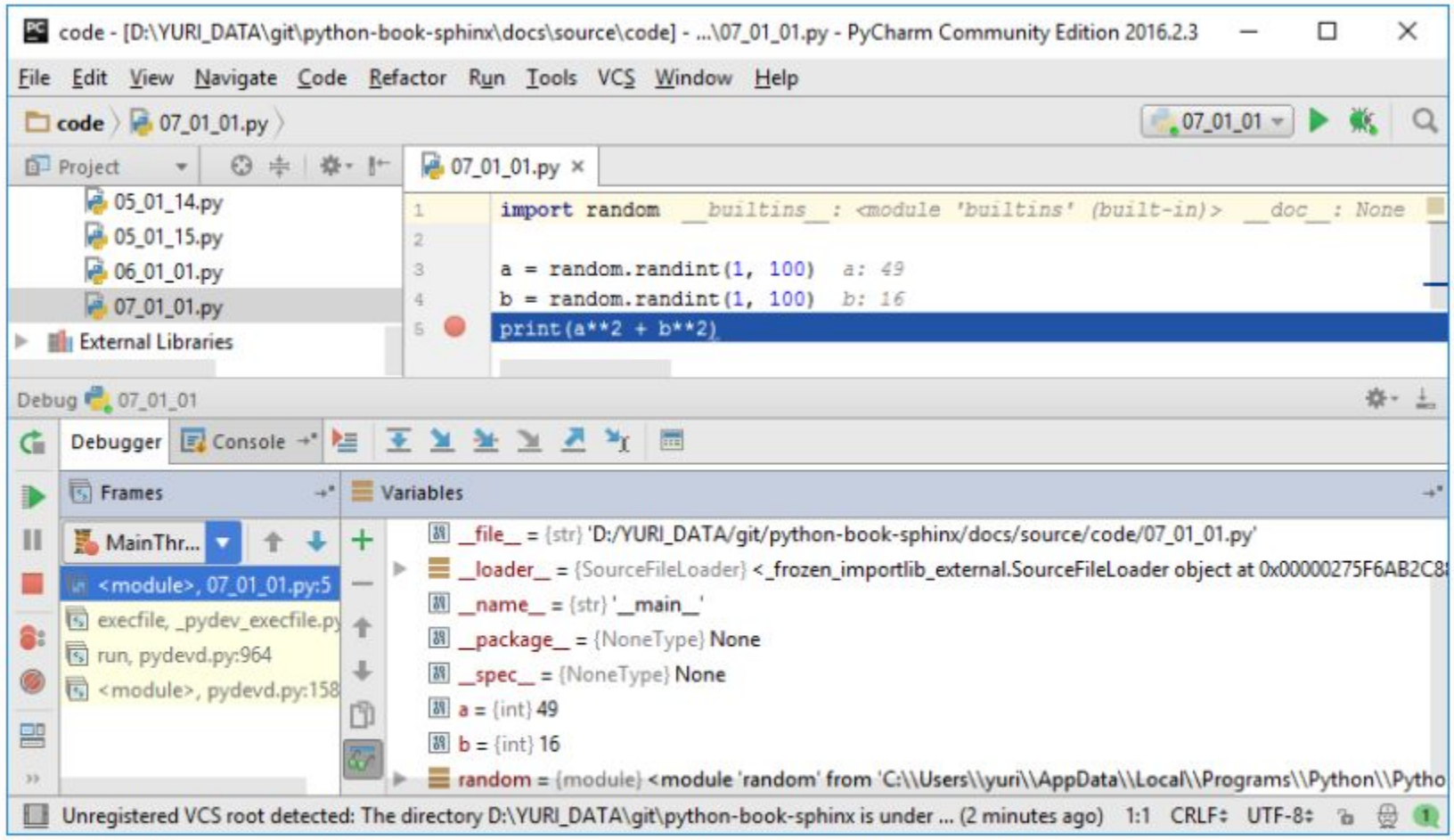
Поиск ошибок и отладка программы

Поиск ошибок выполняется на этапе тестирования программного обеспечения, в процессе которого происходит обнаружение, локализация и устранение ошибок - или отладка. В процессе отладки происходит определение текущих значений переменных и путей выполнения программы, приведших к сбою.

Существуют две взаимодополняющие технологии отладки:

- использование отладчиков: программ, которые включают в себя пользовательский интерфейс для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении определенного условия

Поиск ошибок и отладка программы



Пример отладки в IDE PyCharm: выполнение «заморожено» на точке останова при этом IDE отображает текущие значения переменных, дополнительные окна и параметры

Поиск ошибок и отладка программы

- вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода — на экран, принтер, громкоговоритель или в файл. Вывод отладочных сведений в файл называется журналированием (также *логгированием* или *аудитом*)

Пример отладочного вывода с использованием функции

```
import random

a = random.randint(1, 100)
b = random.randint(1, 100)

print(a, b)  # 44 97

print(a**2 + b**2)  # 11345
```

Понятие исключения

При возникновении ошибки времени выполнения Python создает специальный объект - исключение, который позволяет однозначно характеризовать возникшую ошибочную ситуацию. Выбор подходящего исключения происходит из встроенной иерархии классов-исключений (фрагмент):

- `BaseException` (базовое исключение)
  - `SystemExit` (исключение, порождаемое функцией `sys.exit()` при выходе из программы)
  - `KeyboardInterrupt` (прерывании программы пользователем, `Ctrl+C`)

Понятие исключения

- `Exception` (базовое несистемное исключение)
  - `ArithmeticError` (арифметическая ошибка)
    - `FloatingPointError` (неудачное выполнение операции с плавающей запятой)
    - `OverflowError` (результат арифметической операции слишком велик для представления)
    - `ZeroDivisionError` (деление на ноль)
  - `LookupError` (некорректный индекс или ключ)
    - `IndexError` (индекс не входит в диапазон элементов)
    - `KeyError` (несуществующий ключ)
  - `MemoryError` (недостаточно памяти)
  - `NameError` (не найдено переменной с таким именем)
  - `OSError` (ошибка, связанная с ОС - есть подклассы, например `FileNotFoundError`)
  - `SyntaxError` (синтаксическая ошибка, включает классы `IndentationError` и `TabError`)
  - `SystemError` (внутренняя ошибка)
  - `TypeError` (операция применена к объекту несоответствующего типа)
  - `ValueError` (аргумент правильного типа, но некорректного значения)

Понятие исключения

Пример встроенного возбуждения исключения:

```
>>> "я - строка" / 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Конструкция try

Придерживаясь идеологии «Легче попросить прощения, чем разрешения», Python предусматривает конструкцию `try` для обработки возникающих исключений.

try

except

else

finally

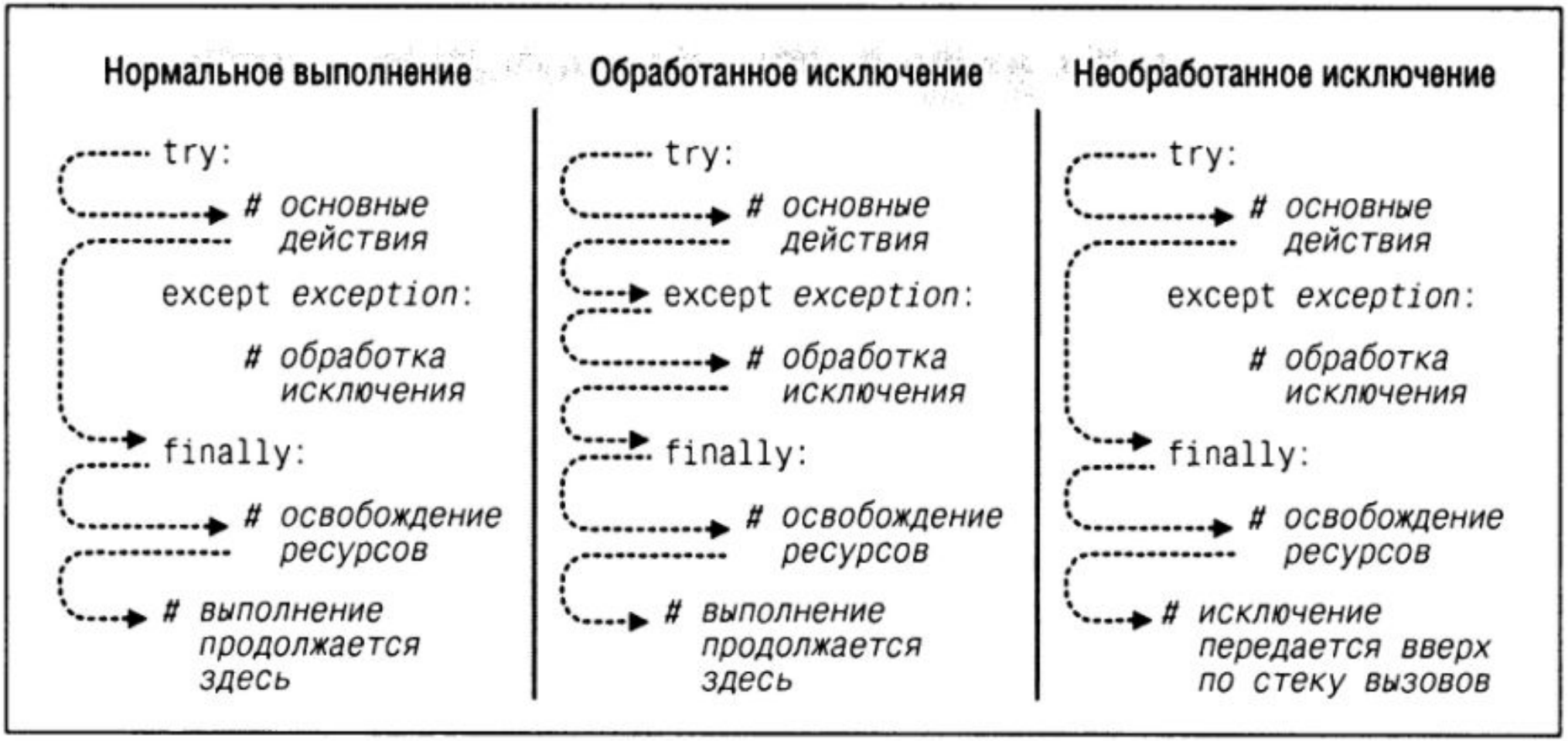
```
try:                                # (try строго 1)
    try_ suite                      # код, который может выполниться с ошибкой
except exception_group1 as var1:    # (except - 0 (если есть finally) и более)
    except_suite1                   # код, выполняемый в случае исключения 'exception_group1'
...                                 # ссылка на исключение может быть записана в 'var1'
except exception_groupN as varN:
    except_suiteN                   # код, выполняемый в случае исключения 'exception_groupN'
...                                 # except-блоков может быть произвольное кол-во
else:                               # (else - 0 или 1)
    else_suite                      # выполняется, если try не завершен преждевременно (например, brea
finally:                            # (finally - 0 или 1)
    finally_suite                   # код, который должен выполнится всегда (была ошибка выше или нет)
```

Конструкция try

Ход выполнения:

- код, который потенциально может привести к ошибке, помещается в блок `try`;
- в случае ошибки, код немедленно завершается и переходит в обработчик `except` (если он указан для соответствующего исключения);
- после поток выполнения переходит к `else` (если исключений не было) и `finally` (в любом случае).

Варианты выполнения потока при обработке исключений



Конструкция try

```
try:
    x = int(input("Введите целое число x (для вычисления 1/x): "))
    res = 1 / x

    print("1/{} = {:.2f}".format(x, res))
except:
    print("Произошла ошибка!")

# --------------
# Примеры вывода:

# Введите целое число x (для вычисления 1/x): 3
# 1/3 = 0.33

# Введите целое число x (для вычисления 1/x): qwerty
# Произошла ошибка!
```

Конструкция try

```
try:
    x = int(input("Введите целое число x (для вычисления 1/x): "))
    res = 1 / x

    print("1/{} = {:.2f}".format(x, res))
except:
    print("Произошла ошибка!")

# --------------
# Примеры вывода:

# Введите целое число x (для вычисления 1/x): 3
# 1/3 = 0.33

# Введите целое число x (для вычисления 1/x): qwerty
# Произошла ошибка!
```

Подобный вариант обработки исключений не рекомендуется, т.к. блок `except` будет перехватывать любое исключение, что не позволит точно определить ошибку в коде. Улучшить код можно, добавив обработку исключения по классу

Конструкция try

```
try:
    x = int(input("Введите целое число x (для вычисления 1/x): "))
    res = 1 / x

    print("1/{} = {:.2f}".format(x, res))
except Exception as err:
    print("Произошла ошибка!")
    print("Тип:", type(err))
    print("Описание:", err)

# --------------
# Примеры вывода:

# Введите целое число x (для вычисления 1/x): 3
# 1/3 = 0.33

# Введите целое число x (для вычисления 1/x): 5.5
# Произошла ошибка!
# Тип: <class 'ValueError'>
# Описание: invalid literal for int() with base 10: '5.5'
```

Конструкция try

```
try:
    x = int(input("Введите целое число x (для вычисления 1/x): "))
    res = 1 / x

    print("1/{} = {:.2f}".format(x, res))
except ZeroDivisionError:
    print("На ноль делить нельзя!")
except ValueError as err:  # 'err' содержит ссылку на исключение
    print("Будьте внимательны:", err)
except (FileExistsError, FileNotFoundError):  # Исключения можно перечислять в виде кортежа
    print("Этого никогда не случится - мы не работаем с файлами")
except Exception as err:
    # Все, что не обработано выше и является потомком 'Exception',
    # будет обработано здесь
    print("Произошла ошибка!")
    print("Тип:", type(err))
    print("Описание:", err)

# --------------
# Примеры вывода:

# Введите целое число x (для вычисления 1/x): 3
# 1/3 = 0.33

# Введите целое число x (для вычисления 1/x): 0
# На ноль делить нельзя!

# Введите целое число x (для вычисления 1/x): qwerty
# Будьте внимательны: invalid literal for int() with base 10: 'qwerty'
```

Конструкция try

```
try:
    x = int(input("Введите целое число x (для вычисления 1/x): "))
    res = 1 / x

    print("1/{} = {:.2f}".format(x, res))
except ZeroDivisionError:
    print("На ноль делить нельзя!")
except ValueError as err:  # 'err' содержит ссылку на исключение
    print("Будьте внимательны:", err)
except (FileExistsError, FileNotFoundError):  # Исключения можно перечислять в виде кортежа
    print("Этого никогда не случится - мы не работаем с файлами")
except Exception as err:
    # Все, что не обработано выше и является потомком 'Exception',
    # будет обработано здесь
    print("Произошла ошибка!")
    print("Тип:", type(err))
    print("Описание:", err)

# --------------
# Примеры вывода:

# Введите целое число x (для вычисления 1/x): 3
# 1/3 = 0.33

# Введите целое число x (для вычисления 1/x): 0
# На ноль делить нельзя!

# Введите целое число x (для вычисления 1/x): qwerty
# Будьте внимательны: invalid literal for int() with base 10: 'qwerty'
```

Теория

Модули и пакеты являются неотъемлемой частью модульного программирования - организации программы как совокупности небольших независимых блоков, структура и поведение которых подчиняются определенным правилам.

Разработка программы как совокупности модулей позволяет:

- упростить задачи проектирования программы и распределения процесса разработки между группами разработчиков;
- предоставить возможность обновления (замены) модуля, без необходимости изменения остальной системы;
- упростить тестирование программы;
- упростить обнаружение ошибок.

Программный код часто разбивается на несколько файлов, каждый из которых используется отдельно от остальных. Одним из методов написания модульных программ является объектно-ориентированное программирование.

Модули

Модуль (*англ.* Module) - специальное средство языка программирования, позволяющее объединить вместе данные и функции и использовать их как одну функционально-законченную единицу (например, математический модуль, содержащий тригонометрические и прочие функции, константы π, ϵ и т.д.).

Пакеты (*англ.* Package) являются еще более крупной единицей и представляют собой набор взаимосвязанных модулей, предназначенных для решения задач определенного класса некоторой предметной области (например, пакет для решения систем уравнений, который может включать математический модуль, модуль со специальными типами данных и т.д.).

Модули

Модуль - отдельный файл с кодом на Python, содержащий функции и данные:

- имеет расширение `*.py` (имя файла является именем модуля);
- может быть импортирован (подключен) (директива `import ...`);
- может быть многократно использован.

Пакеты в Python - это способ структуризации модулей. Пакет представляет собой папку, в которой содержатся модули и другие пакеты и обязательный файл `__init.py__`, отвечающий за инициализацию пакета.

Модули

Так, например, пакет xml имеет следующую структуру:

```
C:\USERS\YURI\APPDATA\LOCAL\PROGRAMS\PYTHON\PYTHON35\LIB\xml
│   __init__.py               Файл инициализации пакета xml
│
├───dom                       Вложенный пакет xml.dom
│       domreg.py
│       expatbuilder.py
│       minicompat.py
│       minidom.py
│       NodeFilter.py
│       pulldom.py
│       xmlbuilder.py
│       __init__.py
│
├───etree                     Вложенный пакет xml.etree
│       cElementTree.py
│       ElementInclude.py
│       ElementPath.py
│       ElementTree.py
│       __init__.py
│
├───parsers                   Вложенный пакет xml.parsers
│       expat.py
│       __init__.py
│
└───sax                       Вложенный пакет xml.sax
        expatreader.py
        handler.py
        saxutils.py
        xmlreader.py
        _exceptions.py
        __init__.py
```

Модули

где каждый модуль (или вложенный пакет) отвечает за свою часть реализации работы с XML-форматом, однако рассматривается как единое целое в виде пакета.

Одна из основных целей использования как модулей, так и пакетов - реализация модели пространства имен, позволяющей логически группировать и в то же время изолировать различные идентификаторы. Например, при наличии глобальной переменной `author` в модуле `A` и `B` не произойдет конфликта, т.к. они находятся в разном пространстве имен: `A.author` и `B.author` соответственно.

Модули

Все модули/пакеты в Python можно разделить на 4 категории:

1. Встроенные (*англ.* Built-in).

   Модули, встроенные в язык и предоставляющие базовые возможности языка (написаны на языке Си).

   К встроенным относятся как модули общего назначения (например, `math` или `random`), так и плаиформозависимые модули (например, модуль `winreg`, предназначенный для работы с реестром ОС Windows, устанавливается только на соответствующей ОС).

Модули

Список установленных встроенных модулей можно посмотреть следующим образом:

```
>>> import sys
>>>
>>> print(sys.builtin_module_names)  # Встроенные модули
('_ast', '_bisect', '_codecs', '_codecs_cn', '_codecs_hk', '_codecs_iso2022',
'_codecs_jp', '_codecs_kr', '_codecs_tw', '_collections', '_csv', '_datetime',
'_functools', '_heapq', '_imp', '_io', '_json', '_locale', '_lsprof', '_md5',
'_multibytecodec', '_opcode', '_operator', '_pickle', '_random', '_sha1',
'_sha256', '_sha512', '_signal', '_sre', '_stat', '_string', '_struct',
'_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', '_winapi',
'array', 'atexit', 'audioop', 'binascii', 'builtins', 'cmath', 'errno',
'faulthandler', 'gc', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt',
'nt', 'parser', 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')
```

Модули

2. Стандартная библиотека (*англ.* Standard Library).

   Модули и пакеты, написанные на Python, предоставляющие расширенные возможности, например, `json` или `os`.

3. Сторонние (*англ.* 3rd Party).

   Модули и пакеты, которые не входят в дистрибутив Python, и могут быть установлены из каталога пакетов Python (*англ.* PyPI - the Python Package Index, более 90.000 пакетов) с помощью утилиты `pip`:

Модули

4. Пользовательские (собственные).

   Модули и пакеты, создаваемые разработчиком.

> **Примечание**
>
> Создание собственных пакетов не рассматривается в рамках настоящего курса.

В собственной программе рекомендуется выполнять импорт именно в таком порядке: от встроенных до собственных модулей/пакетов.

Модули

Для использования модуля или пакета в коде необходимо его предварительно подключить (импортировать).

import

Импорт модуля или пакета выполняется единожды инструкцией `import`, располагаемой, как правило, в начале файла.

Модули

Выполнить подключение модуля можно несколькими способами:

```
# Способ №1
# В данном способе обратиться к члену модуля можно с указанием модуля,
# например, module_1.object_1

# Импортирует модуль 'module_1'
import module_1

# Импортирует модули 'module_1', 'module_2',..., 'module_n'
import module_1, module_2, ..., module_n

# Импортирует модуль 'module_1' под псевдонимом 'preferred_name'
import module_1 as preferred_name

# Способ №2
# В данном способе обратиться к члену модуля можно без указания модуля,
# например, object_1

# Импортирует 'object' под псевдонимом 'preferred_name' из модуля 'module_1'
from module_1 import object as preferred_name

# Импортирует объекты 'object_1', ..., 'object_n' из модуля 'module_1'
from module_1 import object_1, object_2, ..., object_n

# Импортирует все объекты из модуля 'module_1'
from module_1 import *
```

Модули

```
from math import sin
import math

# Доступ к функции sin() возможен без указания модуля, а
# к функции radians() только с указанием
#
# Данный способ рекомендуется использовать:
# - для повышения читаемости кода;
# - если вероятность конфликта имен с другими участками кода мала.

print("sin(30) = {:.1f}".format(sin(math.radians(30))))  # sin(30) = 0.5
```

Область поиска

При импорте модуля или пакета Python выполняет его поиск в следующем порядке:

1. Встроенный модуль?
2. Файл `module.py` / `module.pyc`, или пакет `package` есть в списке путей переменной `sys.path`?

   `sys.path` - список, при инициализации включающий:

   - рабочую директорию скрипта (основного модуля);
   - переменную окружения `PYTHONPATH` и пути инсталляции Python.

Если модуль не удается найти, возбуждается исключение `ModuleNotFoundError`. При ошибке загрузки существующего модуля - `ImportError`.

Специальные атрибуты

Каждый модуль имеет специальные и дополнительные атрибуты.

1. Специальные атрибуты содержат системную информацию о модуле (путь запуска, имя модуля и др.) и доступны всегда. Некоторые из них:

   `__name__`

   Полное имя модуля.

   Пример: `"math"` или `"os.path"`.

   `__doc__`

   Строка документации.

   `__file__`

   Полный путь к файлу, из которого модуль был создан (загружен).

   Пример: `C:\code\task_09_01_02\fibonacci.py`.

Собственные модули

Python предоставляет возможность создания собственных модулей.

Примечание

PEP8.

Соглашение рекомендует использовать:

- змеиный_регистр (*англ.* snake_case) для наименования модулей: `my_module`;
- строки документации.

Для создания собственных модулей нет особенных правил - любой файл с расширением `*.py` является модулем.

Собственные модули

```
""" Модуль для работы с числами Фибоначчи.

- список чисел Фибоначчи, не превышающих 'n';
- проверка на вхождение в ряд Фибоначчи;
- ...
"""


def list_le_than(n):
    """Вернуть список чисел Фибоначчи, не превышающих 'n'."""
    assert isinstance(n, int) and n > 0

    a, b = 1, 1
    result = [a]
    while b <= n:
        result.append(b)
        a, b = b, a + b
    return result


def is_in_row(n):
    """Вернуть True, если 'n' - число Фибоначчи."""
    return n in list_le_than(n)
```

Собственные модули

```python
import fibonacci

num = 20
print("Числа до {}: {}".format(num, fibonacci.list_le_than(num)))  # [1, 1, 2, 3, 5, 8, 13]
print("{} входит: {}".format(num, fibonacci.is_in_row(num)))  # False
```

Примечание

Основной модуль программы, как правило, удобно называть `main.py`.

Собственные модули

```python
import fibonacci

num = 20
print("Числа до {}: {}".format(num, fibonacci.list_le_than(num)))  # [1, 1, 2, 3, 5, 8, 13]
print("{} входит: {}".format(num, fibonacci.is_in_row(num)))  # False
```

Примечание

Основной модуль программы, как правило, удобно называть `main.py`.