Python

Исключения, копии, декораторы, форматирование

Errors and Exceptions

Синтаксическая ошибка (**Syntax Error**) – ошибка парсинга

```
>>> while True print('Hello world')
    File "<stdin>", line 1, in ?
      while True print('Hello world')
                  ^
SyntaxError: invalid syntax
```

Даже если фраза синтаксически правильна, может возникнуть ошибка во время исполнения – это исключение (**Exception**)

```
>>> 10 * (1/0)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

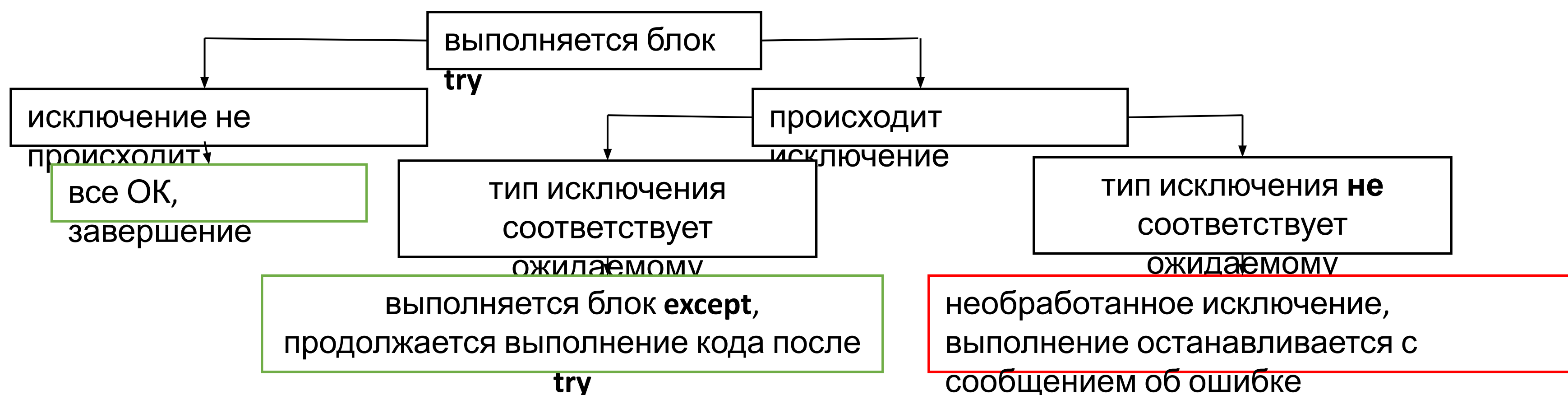
тип исключения

причина исключения

Errors and Exceptions

Перехват исключений
(**Handling Exceptions**)

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number. Try again...")
...
```



Errors and Exceptions

Перехват нескольких типов исключений

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an
integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Блоки **else** и **finally**

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        # Если сработал try и не сработал except
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
    finally:
        # Выполняется в любом случае
        print('Goodbye, world!')
```

Errors and Exceptions

Вызов исключений

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
NameError: HiThere
```

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
    File "<stdin>", line 2, in ?
NameError: HiThere
```

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and
'str'
```

Копии

Создание копии

```
>>> x = [1, 2, 3]
>>> y = x
>>> y[2] = 5
>>> y
[1, 2, 5]
>>> x
[1, 2, 5]
```

Правильный

```
>>> x = [1,2,3]
>>> y = x[:]
>>> y.pop()
3
>>> y
[1, 2]
>>> x
[1, 2, 3]
```

Копирование вложенных

```
import copy

my_dict = {'a': [1, 2, 3], 'b': [4, 5, 6]}
my_copy_dict = copy.deepcopy(my_dict)
```

Форматирование строк

Стандартный оператор %

```
>>> name = 'Reuven'
>>> "Hello, %s" % name
'Hello, Reuven'
```

```
>>> first = 'Reuven'
>>> last = 'Lerner'
>>> "Good morning, %s %s" % (first, last)
'Good morning, Reuven Lerner'
```

Оператор str.format

```
>>> "Good morning, {} {}".format(first, last)
'Good morning, Reuven Lerner'
```

```
>>> "Good morning, {1} {0}".format(first, last)
'Good morning, Lerner Reuven'
```

нумерованные аргументы

```
>>> names = ('Reuven', 'Lerner')
>>> "Good morning, {} {}".format(*names)
'Good morning, Reuven Lerner'
```

список аргументов

```
>>> "Good morning, {first} {last}".format(first='Reuven',
last='Lerner')
'Good morning, Reuven Lerner'
```

именованные аргументы

```
>>> person = {'first':'Reuven', 'last':'Lerner'}
>>> "Good morning, {first} {last}".format(**person)
'Good morning, Reuven Lerner'
```

словарь аргументов

Форматирование строк

```
>>> person = {'first':'Reuven', 'last':'Lerner'}
>>> "Good {0}, {first} {last}".format('morning', **person)
'Good morning, Reuven Lerner'
```

смешанные аргументы (**не рекомендуется!**)

```
>>> "Your name is {name:10}".format(name="Reuven")
'Your name is Reuven    '
```

указание количества подставляемых символов

```
>>> "Your name is {name:>10}".format(name="Reuven")
'Your name is     Reuven'
```

выравнивание по правой стороне (по левой ‘<’)

```
>>> "Your name is
{name:*^10}".format(name="Reuven")
'Your name is **Reuven**'
```

выравнивание по центру

```
>>> "The price is ${number}.".format(number=123)
'The price is $123.'
```

подстановка чисел

```
>>> "The price is ${number:b}.".format(number=5)
'The price is $101.'
```

подстановка числа в двоичном виде

```
>>> "The price is ${number:x}.".format(number=123)
'The price is $7b.'
```

подстановка числа в шестнадцатеричном виде

```
>>> "Your call is important to us. You are call
#{number:05}.".format(number=123)
'Your call is important to us. You are call #00123.'
```

дополнение лидирующими нулями

Форматирование строк

Внутри {} нельзя использовать исполняемый python-код — вместо этого предлагается простенький микроязык, отдельный и отличающийся от python в целом. НО:

1. можно получить значения атрибутов/свойств

```
>>> class Foo(object):
        def __init__(self):
            self.x = 100
>>> f = Foo()
>>> 'Your number is {o.x}'.format(o=f)
'Your number is 100'n
```

(получить атрибут можно, а вот выполнить его —

```
>>> "Your name is {name.upper()}".format(name="Reuven")
AttributeError: 'str' object has no attribute 'upper()'
```

2. можно взять элемент итерируемого

```
>>> "Your favorite number is {n[3]}.".format(n=numbers) 'Your favorite number is
```

(но операции разрезания (**slice**) не поддерживаются)

```
>>> "Your favorite numbers are {n[2:4]}.".format(n=numbers) ValueError: Missing ']' in format string
```

3. можно использовать [] и для получения записей в словаре по имени, но имя вводится без кавычек

```
>>> person = {'first':'Reuven', 'last':'Lerner'}
>>> "Your name is {p[first]}.".format(p=person)
'Your name is Reuven.'
```

```
>>> "Your name is {p['first']}.".format(p=person)
KeyError: "'first'"
```

Декораторы

Декоратор – функция, которая может изменять поведение другой функции

```
def bold(fun_hello):
    def inner(who):
        print "<b>"
        fun_hello(who)
        print "</b>"
    return inner
```

декорирующая функция

```
@bold
def hello(who):
    print "Hello", who
```

использование декоратора

```
>>> hello("World")
>>> # bold(hello)("World")
<b>
Hello World
</b>
```

равносильно вызову

результат

```
def italic(fun_hello):
    def inner(who):
        print "<i>"
        fun_hello(who)
        print "</i>"
    return inner
```

```
@italic
@bold
def hello(who):
    print "Hello", who
<i>
<b>
Hello World
</b>
</i>
```

несколько декораторов

Декораторы

Декоратор с

```
def tag(name):
    def decorator(fun_hello):
        def inner(who):
            print "<%s>" % name
            fun_hello(who)
            print "</%s>" % name
        return inner
    return decorator

@tag("b")
def hello(who):
    print "Hello", who
```

```
>>> hello("World")
<b>
Hello World
</b>
```

Вежливые декораторы

сохранение названия функции после

```
>>> hello.__name__
'inner'
```

```
def bold(fun_hello):
    def inner(who):
        print "<b>"
        fun_hello(who)
        print "</b>"
    inner.__name__ = fun_hello.__name__
    return inner
```

```
>>> hello.__name__
'hello'
```

Чтобы вручную не сохранять __name__, __module__ и __doc__ можно воспользоваться стандартым декоратором

```
from functools import wraps
def bold(fun_hello):
    @wraps(fun_hello)
    def inner(who):
        . . .
    return inner
```

Декораторы

Шаблоны декораторов

Декоратор без

```
from functools import wraps

def название_декоратора(декорируемая_функция):
    @wraps(декорируемая_функция)
    def inner(параметры_декорируемой_функции):
        ...
        декорируемая_функция(параметры_декорируемой_функции)
        ...
    return inner
```

Декоратор с

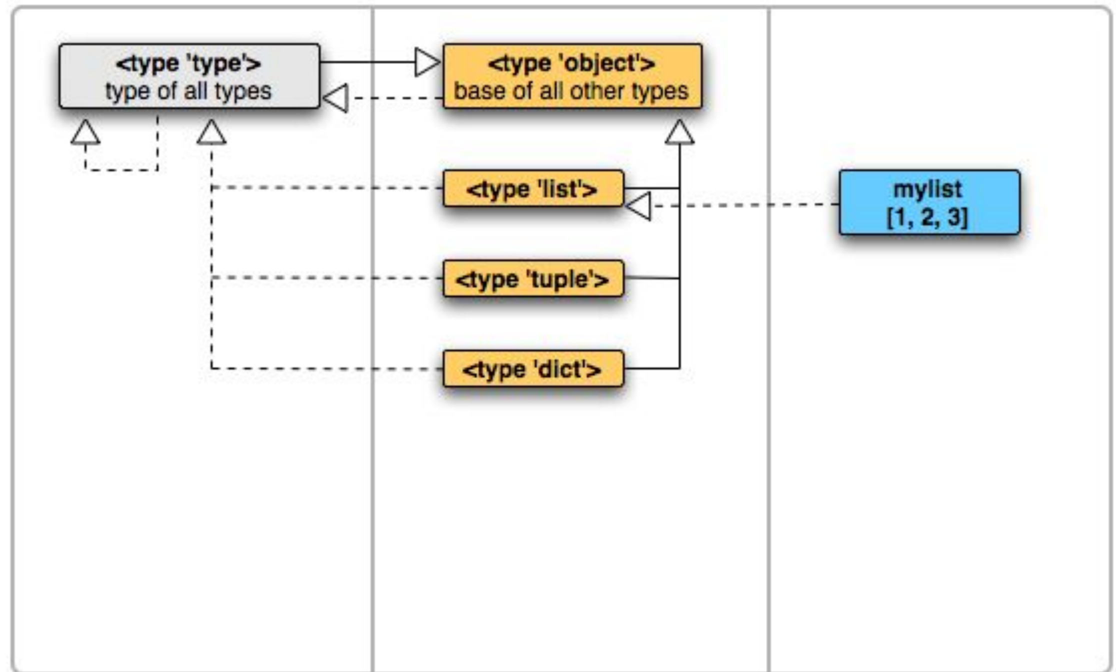
```
from functools import wraps

def название_декоратора(параметры_декоратора):
    def decorator(декорируемая_функция):
        @wraps(декорируемая_функция)
        def inner(параметры_декорируемой_функции):
            ...
            декорируемая_функция(параметры_декорируемой_функции)
            ...
        return inner
    return decorator
```

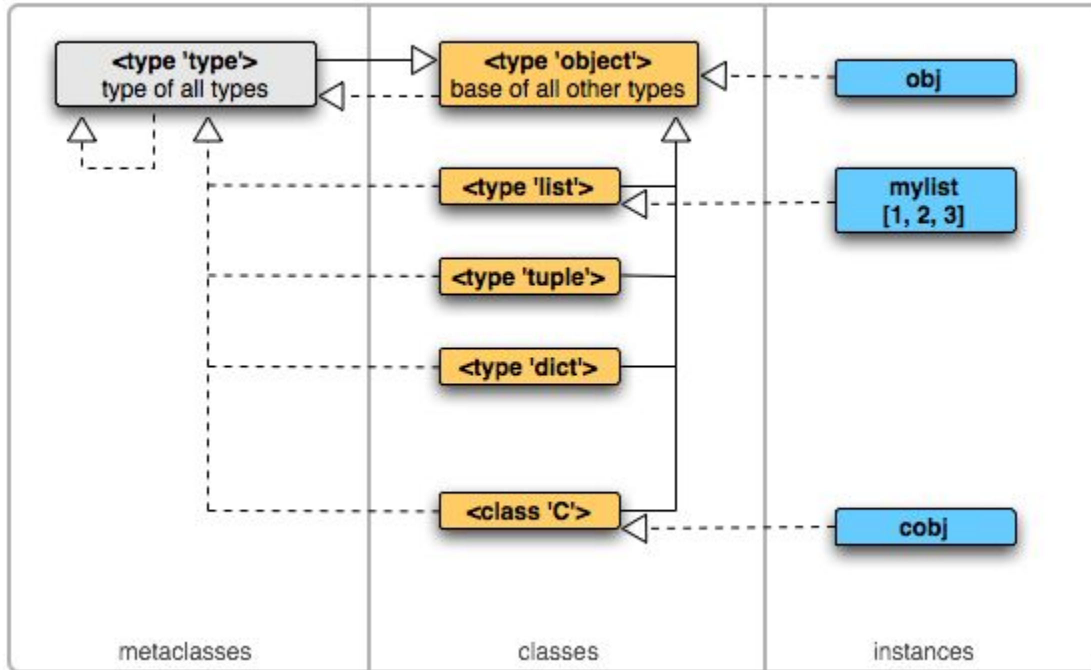
Объекты в Python

Class is Type is Class

```
>>> object
<type 'object'>
>>> type
<type 'type'>
>>> type(object)
<type 'type'>
>>> object.__class__
<type 'type'>
>>> object.__bases__
()
>>> type.__class__
<type 'type'>
>>> type.__bases__
(<type 'object'>,)
```



The Python Objects Map



Пользовательские классы

```
>>> class A(object):
...     qux = 'A'
...     def __init__(self, name):
...         self.name=name
...     def foo(self):
...         print 'foo'
...
>>> a = A('a')
```

```
>>> a.__dict__    {'name': 'a'}
>>> a.__class__
<class '__main__.A'>
>>> type(a)
<class '__main__.A'>
>>> a.__class__ is type(a)
True
```

```
>>> class A(object):
...     pass
...

>>> isinstance(A, object)
True


>>> a = A()
>>> isinstance(a, A)
True
>>> isinstance(a, object)
True
>>> isinstance(a, type)
False
```