Компьютерные технологии в математическом моделировании

Лекция 2.

Ассистент кафедры
математической физики
к.ф.-м.н.
Татьяна Евгеньевна Романенко

Содержание лекции

- Функции
- Исключения
- Модули

Функции

- Определение:

```
def <name>(arg1, arg2,... argN):
        code_block
        return <value>
```

- Пример:

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
# пример вызова
fib(20)
```

Функции: области определения

- Лексические области видимости:
  - локальные переменные (определены внутри инструкции def)
  - нелокальные для данной функции (определены в объемлющем def)
  - глобальные переменные (за пределами всех def)

- Объемлющий модуль – глобальная область видимости (атрибуты объекта модуля вовне и простые переменные внутри)
- Глобальная область видимости охватывает один файл
- Каждый вызов функции создает новую локальную область видимости
- Операции присваивания создает локальные имена, если они не были объявлены нелокальными или глобальными
- Операции изменения не рассматривают имена как локальные
- Правило разрешения имен: LEGB (local, enclosing, global, built-in)
- Фабричная функция/замыкание

Фабричная функция/замыкание

```
def  make_pow(N):
        def  action(X):
                return X ** N
        return action


f  = make_pow(2)
g = make_pow(3)
x = f(3)  # x = 9
y = f(5)  # y = 25
z = g(2) # z = 8
```

- Вложенная функция продолжает хранение переданное значение N, хотя объемлющая функция уже завершилась

Значения по умолчанию

- Сохранение состояния объемлющей области видимости с помощью аргументов по умолчанию

```
x = 8
def f(x=x):
    #...
```

Использование nonlocal и global

```
def test_nonlocal(start_value):
    state = start_value
    def nested_func(message):
        nonlocal state
        print(message, state)
        state += 1
    return nested_func
```

```
F = test_nonlocal(0)
F('hello')          # hello 0
F('hello2')         # hello2 1
```

Сохранение информации в атрибутах функции

```
def test_attributes(start_value):
    def nested_func(message):
        print(message, nested_func.state)
        nested_func.state += 1
    nested_func.state = start_value
    return nested_func
```

```
G = test_attributes(0)
G('hello')          # hello 0
G('hello2')         # hello2 1
G.state = 10
G('hello again')    # hello again 10
```

Передача аргументов

- Аргументы передаются через автоматическое присваивание объектов локальным переменным
- Операция присваивания именам аргументов внутри функции не оказывает влияния на вызывающую программу
- Изменение внутри функции аргумента, который является изменяемым объектом, может оказать влияние на вызывающую программу
- Неизменяемые объекты передаются «по значению»
- Изменяемые объекты передаются «по ссылке»

```
def   change_arguments(a, b):
    a = 3
    b[0] = 'hello'
```

```
X = 1
Y = [1, 'hi', 118.0]
change_arguments(X, Y)
# X = 1, Y = [‘hello’, ‘hi’, 118.0]
```

Как избежать воздействия на изменяемые аргументы

- Передача копии аргумента в вызове функции

```
X = 1
Y = [1, 'hi', 118.0]
change_arguments(X, Y[:])
# X = 1, Y = [1, ‘hi’, 118.0]
```

- Создание копии внутри функции

```
def   change_arguments(a, b):
    b = b[:]
    a = 3
    b[0] = 'hello'
```

Специальные режимы сопоставления аргументов

- Сопоставление по позиции (значения и имена ставятся в соответствие по порядку, слева направо)
- Сопоставление по именам
- Значения по умолчанию
- Переменное число аргументов (прием произвольного числа аргументов * и **)
- Переменное число аргументов (передача произвольного числа аргументов *)
- Только именованные аргументы

- В вызывающей программе:
  - Обычный аргумент(сопоставление по позиции) func(value)
  - Именованный аргумента(сопоставление по указанному имени) func(name=value)
  - Все объекты последовательности передаются как отдельные позиционные аргументы func(*sequence)
  - Все пары ключ\значение передаются как отдельные именованные аргументы func(**dict)

В функции:

- Обычный аргумент(сопоставление по позиции или имени) `def func(name)`
- Значение аргумента по умолчанию (если он не передается в функцию) `def func(name=value)`
- Определяет и объединяет все дополнительные объекты в кортеж `def func(*name)`
- Определяет и объединяет все дополнительные именованные объекты в словарь `def func(**name)`
- Аргументы, которые должны передаваться в функцию только по именам `def func(*args, name)` `def func(*, name=value)`

Особенности сопоставления

- В вызове функции:
  - Любые позиционные аргументы(значения)
  - Любые именованные аргументы
  - Аргументы в форме *sequence
  - Аргументы в форме **dict

- В заголовке функции:
  - Любые обычные аргументы
  - Аргументы со значениями по умолчанию
  - Аргументы в форме *name \ *
  - Любые имена или пары name=value, которые передаются только по имени
  - Аргументы в форме **name

Порядок действий интерпретатора при сопоставлении аргументов

- Сопоставление неименованных аргументов по позициям
- Сопоставлением именованных аргументов по именам
- Сопоставление дополнительных неименованных аргументов с кортежем *name
- Сопоставление дополнительных именованных аргументов со словарем **name
- Сопоставление значений по умолчанию с отсутствующими именованными аргументами

Примеры фиксированного числа аргументов

```
def print_info(name, age, job, city = 'Moscow', university =
    'MSU'):
    print (name, age, job, city, university)
print_info('Peter', 28, 'team lead')
print_info(name = 'Alex', job = 'developer', age = 31)
print_info(name = 'Pavel', city = 'St Petersburg', age = 21, job =
    'designer')
print_info('Sergey', 41, 'scientist', university = 'MGTU')
```

```
Peter 28 team lead Moscow MSU
Alex 31 developer Moscow MSU
Pavel 21 designer St Petersburg MSU
Sergey 41 scientist Moscow MGTU
```

Примеры произвольного числа аргументов

```
def print_info2(age, *names, **jobs):
    print(age)
    print(names)
    print(jobs)
print_info2(31, 'Erich', 'Maria', 'Remarque', Military='Soldier',
    Civil = 'Novelist')
```

```
31
('Erich', 'Maria', 'Remarque')
{'Military': 'Soldier', 'Civil': 'Novelist'}
```

Аргументы, передающиеся только по именам-1

```
def  named_only(age, *names, job):
    print(age)
    print(names)
    print(job)

named_only(30, 'Erich', 'Maria', 'Remarque', job = 'Novelist')
```

```
30
('Erich', 'Maria', 'Remarque')
Novelist
```

Аргументы, передающиеся только по именам-2

```
def named_only2(age, *, name, job):
    print(age)
    print(name)
    print(job)



named_only2(30, name='Erich Maria Remarque',
   job='Novelist')
```

```
30
Erich Maria Remarque
Novelist
```

Типичные ошибки при работе с функциями

- Локальные имена определяются статически

```
def func():
    print(X)
    X = 5
```

```
Traceback (most recent call last):
...
UnboundLocalError: local variable 'X' referenced before assignment
```

- Значения по умолчанию и изменяемые объекты (значения по умолчанию сохраняются между вызовами функции)

```
def func(x=[]):
    x.append(1)
    print(x)
func([2])
func()
func()
```

```
[2, 1]
[1]
[1, 1]
```

- Функции, не возвращающие результат

```
L = [1, 2, 3]
L = L.append(4)
print(L)
```

```
None
```

Рекурсивные функции

```
from sys import setrecursionlimit
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
#print(factorial(1005))
setrecursionlimit(10000)
print(factorial(1005))
```

120

RecursionError: maximum recursion depth exceeded in comparison

4084573625427926461960411776755798447068222734620727479639565495066604130413823740779170
1570663340627045936259028703253337249414349109419306961297460440826907927883843595469223
9143722955717545999882172106976837006382326754472538839760172730837588374373142640222890
3709475327186525689108132608784955033064028816259485533853052367975887427098785294399137
3739545306794396038301119514332644236031815966499812482708623864545219950566544061143070
0504774354836638975180596983963491529399217533660711708126182777236351926183015536451166
0370722211394367156071388657777299560331619493861205958913147103992613172710308459642746
9818092522014601694033063600802591218925659987326975988933236160576652958735234319068671
6983743889385840824094385163071387935697239250276274086421457499910969899232159703894869
5482568243040350973338501824885503656373470920540950928891846661711086294925449559752832
7542725184767625513779553983710330488246882938090902886623855560488071421388303650864773
9174433955876881072906113196381132622374988738640863569931701169799545699650749772450080
0696167009948653792475056767499455003568512086321140590754889632503742306247775038252681
3442573795079434208540899239460429387384410881972637760693875101693136716924307081517265
2134649044805717671690375916247717978135659901857204091866873403680767627937198516862811
7551657680807858814290258934858398880821285938179951763011348240614294171024226279234587
5716075433530815759809734655736208960331682820470658969511063265031342085409273392084512
9987759360531180697374530176239339196381612546544061438362820181158695103192478238259557
9216596782798819033799719517762886415046891379245611528021946574239598138537655376529295
0261962825214473486170996687232985631064650788115959658206355667318731078502877724104267
6198832003978529162119280203597561110752417715128202488276666405742614501522306432262295
7050287473462549813746361031163547817886371379853706943773297360377007293413478532214026
7520464751063152128104061542302033705969644491281435028217715617070927736555454686243367
4001703112748870231340066766133676021702118698810913316613093623234453383461478268506370
0283984248291222567176523481168919215305157624444005819898305998288518910152418499056438
5625351059834179666738406813247524927630978635331042454633281868341338000008078531317551
9808813489256363260843268417244736589817446400000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000

Анонимные функции: lambda

lambda arg1, arg2,..., argN: выражение,
использующее аргументы

- lambda – выражение, а не инструкция (может быть внутри литералов или вызовов функций)
- Тело lambda это не блок инструкций, а единственное выражение

```
f = lambda x, y, z: x + y + z
print(f(2,3,4))
```

9

Пример lambda-функции

```
def print_knight_info():
    title = 'Sir'
    action = lambda x: title + ' ' + x
    return action

act = print_knight_info()
print(act('Lancelot'))
```

Sir Lancelot

Обработка исключений

```
try:
   # main part
except  Exception1:
   # …
except  (Exception2, Exception3):
   # …
except  type as value:
   # …

…
else:
   # else part
finally:
   # finally part
```

- Инструкция raise

```
raise
raise exc
raise ExceptionClass
raise exc from exc2
```

Finally:

- в main возникло исключение и оно было обработано
- в main возникло исключение и оно не было обработан
- в main не возникло исключения
- в одном из обработчиков возникло новое исключение

Пример работы с исключениями

```
L = [1, 2, 'ab', 17.0]
L = [1, 2, 34.0]
try:
    for elem in L:
        print(sin(elem))
except (ValueError, TypeError) as inst:
    print(type(inst))
    print(inst)
else:
    print('Everything was ok!')
finally:
    print('We finally got here!')
```

```
0.8414709848078965
0.9092974268256817
<class 'TypeError'>
a float is required
We finally got here!
```

```
0.8414709848078965
0.9092974268256817
0.5290826861200238
Everything was ok!
We finally got here!
```

Создание исключений

- Инструкция raise

```
raise
raise exc
raise ExceptionClass
raise exc from exc2
```

- Инструкция assert

```
assert  test, data
```

```
def f(x):
    assert x < 0, 'x must be negative'
    return x ** 2

f(1)
```

```
Traceback (most recent call last):
…
    assert x < 0, 'x must be negative'
```

Иерархия исключений

- BaseException
- Exception
- ArithmeticException
- OverflowException
- RuntimeError
- SystemError
- …

Модули

- Повторное использование кода
- Разделение системы пространства имен
- Реализация служб или данных для совместного использования

- Структура программы:
  - Основной выполняемый файл\сценарий
  - Подключаемые модули

- Как работает import
  - Поиска файла модуля
  - Компиляция(если необходимо) в байт-код
  - Запуск кода модуля, чтобы создать объекты, которые он определяет

Путь поиска модулей

- Домашний каталог программы
- Содержимое переменной окружения PYTHONPATH (если она определена)
- Каталоги стандартной библиотеки
- Содержимое любых файлов с расширением .pht

```
import sys
print(sys.path)
```

Поиск идет по имени:

- Файл с исходным текстом (.py)
- Файл с байт-кодом (.pyc)
- Скомпилированный модуль расширения на C\C++ (.so, .dll, .pyd)
- Скомпилированный встроенный модуль на C, статически скомпонованный с интерпретатором Python
- Класс Java для Jython
- Компонент .NET в версии IronPython

Использование модуля

- Инструкция import m
- Инструкция from m import f1, f2
- Инструкция from m import *

- Импорт выполняется только 1 раз
- import и from – операции присваивания
- Изменение значений имен в других файлах

```
% python
>>> from small import x, y      # Скопировать два имени
>>> x = 42                      # Изменить только локальное имя x

>>> import small                # Получить имя модуля
>>> small.x = 42                # Изменить x в другом модуле
```

- Импорт одноименных функций модуля

Пространства имен модулей

- Инструкции модуля выполняются во время 1 попытки импорта
- Операции присваивания, выполняемые на верхнем уровне, создают атрибуты модуля
- Доступ к пространствам имен модуля через атрибут __dict__ или dir(M)
- Модуль – единая область видимости (локальная является глобальной)
- Операция импортирования не меняет областей видимости:
  - Функциям недоступны имена из других функций (кроме вложенных)
  - Имена из других модулей, кроме явно импортированных, недоступны

Квалификация имен видимости

- Простые переменные
  - Использование краткой формы имени (X) – правило LEGB
- Квалифицированные имена
  - X.Y – поиск имени X в текущей области, затем поиск атрибута Y в объекте X
- Квалифицированные пути
  - X.Y.Z – поиск имени Y в объекте X, затем поиск имения Z в объекте Y
- Общий случай
  - Квалификация имен применима ко всем объектам, имеющим атрибуты: модули, классы, расширения типов на C и т.п.

Пространства имен модулей

```
import testmodule
from imp import reload
print(testmodule.sys)
print(testmodule.result)
print(testmodule.func)

print(testmodule.__dict__.keys())

print(testmodule.sys.api_version)
reload(testmodule)
```

```
print('Loading module...')
import sys
result = 42

def func():
    print('Inside testmodule.func() function...')

print('Loading completed.')
```

```
Loading module...
Loading completed.
<module 'sys' (built-in)>
42
<function func at 0x0000007CABCE1E18>
dict_keys(['__builtins__', '__loader__', '__cached__', '__name__', '__doc__', '__file__',
'__spec__', '__package__', 'sys', 'result', 'func'])
1013
Loading module...
Loading completed.
```

Особенности reload

- reload запускает новый программный код в файле модуля в текущем пространстве имен модуля
  - Перезаписывание текущего пространства имен, а не создание\удаление
- Инструкции присваивания на верхнем уровне файла замещают имена новыми значениями
- Повторная загрузка оказывает воздействие на всех клиентов, использовавших инструкцию import для получения доступа к модулю
  - Клиенты, использовавшие полные имена, получат новые атрибуты после перезагрузки
- Повторная загрузка не действует на клиентов, использующих инструкции from
  - Они ссылаются на старые атрибуты

Генераторы списков

```
a = {1:10, 2:20, 3:30}
b = [[i,a[i]] for i in a]
c = [j for i in b for j in i]

d = []
for i in b:
    for j in i:
        d.append(j)

f = [i for i in range(30,250) if i%30 == 0 or i%31 == 0]

# c = d = [1, 10, 2, 20, 3, 30]
# f = [30, 31, 60, 62, 90, 93, 120, 124, 150, 155, 180, 186, 210, 217,
   240, 248]
```

Генераторы

- Генерация последовательности значений с течением времени
- Автоматическая поддержка протокола итераций
- Генератор *поставляет* значение, а не *возвращает* его (выражение yeild)
- Выигрыш по производительности и памяти

```
def gen_squares(N):
    for i in range(N):
        yield i**2

for i in gen_squares(5):
    print(i)
```

```
x = gen_squares(5)
print(x.__next__())
print(x.__next__())
print(next(x))
```

```
0
1
4
9
16
0
1
4
```

Генераторы: метод send()

```
def  generator_send():
    print("send demo")
    index = 1;
    while True:
        message = yield index
        print(message + '    ' + str(index))
        index += 1


demo_generator = generator_send()
print(next(demo_generator))
print(demo_generator.send('First'))
print(demo_generator.send('Second')))
```

```
send demo
1
First 1
2
Second 2
3
```

Выражения-генераторы

```
list1 = [x ** 2 for x in range(4)]
print(list1)

gen = (x ** 2 for x in range(4))
list2 = list(gen)
print(list2)

print(next(gen))
```

```
[0, 1, 4, 9]
[0, 1, 4, 9]
Traceback (most recent call last):
…
StopIteration
```