

# Programming

## Programming & Python Basics

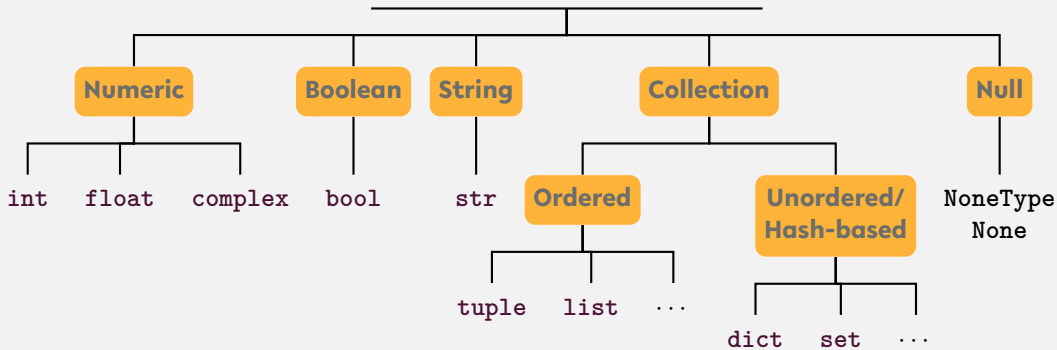
Daniel Dörr

Faculty of Technology, Bielefeld University

```
332
333
334     if extrapolate is None:
335         extrapolate = self.extrapolate
336     x = np.asarray(x)
337     x_shape, x_ndim = x.shape, x.ndim
338     x = np.ascontiguousarray(x.ravel(), dtype=np
339
340     # With periodic extrapolation we map x to the
341     # [self.t[k], self.t[n]].
342     if extrapolate == 'periodic':
343         n = self.t.size - self.k - 1
344         x = self.t[self.k] + (x - self.t[self.k]) *
345         extrapolate = False
346
347     out = np.empty((len(x), prod(self.c.shape[1:])),
348                   dtype=self._ensure_c_contiguous())
349     self._evaluate(x, nu, extrapolate, out)
350     out = out.reshape(x_shape + self.c.shape[1:])
351     if self.axis != 0:
352         # transpose to move the calculated values to t
353         l = list(range(out.ndim))
354         l = l[x_ndim:x_ndim+self.axis] + l[:x_ndim] + l[x_ndim+self.axis:]
355         out = out.transpose(l)
356     return out
357
358 def _evaluate(self, xp, nu, extrapolate, out):
359     _bspl.evaluate_spline(self.t, self.c, reshape(self.c,
360     self.k, xp, nu, extrapolate, out)
361
362 def _ensure_c_contiguous(self):
363     """
364     c and t may be modified by the user. The Cython code
365     ensures that they are C contiguous.
366     """
367     if not self.c.flags.c_contiguous:
368         self.c = np.ascontiguousarray(self.c)
369     if not self.t.flags.c_contiguous:
370         self.t = np.ascontiguousarray(self.t)
```

# Recap

# Python Data Types



... and user-defined types

# String

## `str()`

- ❖ instantiation: `s = 'a new string'` or `s = "a new string"`
- ❖ length: `len(s)`
- ❖ access:
  - ❖ first: `s[0]`
  - ❖ slice: `s[1:3]`
  - ❖ last: `s[-1]`
- ❖ existence: `'n' in s` or `'new' in s`
- ❖ frequency: `s.count('new')`

# List

## `list()`

- ❖ instantiation: `l = [1, 2, 3]`
- ❖ length: `len(l)`
- ❖ add elements: `l.append("content")`
- ❖ access:
  - ❖ first: `l[0]`
  - ❖ slice: `l[1:3]`
  - ❖ last: `l[-1]`
- ❖ existence: `2 in l`
- ❖ location: `l.index(3)`

# Complex data: Mappings

## `dict()`

- ❖ instantiation: `d = dict()`, `d = {'x': 1, 'y': 2 }, ...`
- ❖ length: `len(l)`
- ❖ add elements: `d['a'] = 'ef'`
- ❖ access: `d['a']`
- ❖ existence: `'a' in d`

# Conditional statements: if/else clause

```
if «Boolean expression»:
```

```
    «statement»
```



Mind the indention!

---

**OR**

---

```
if «Boolean expression»:
```

```
    «statement»
```

```
else:
```

```
    «alternative statement»
```

# Boolean operators, Comparisons

- Elementary logic: `and`, `or`, `not`
- Comparators:
  - `==` “is equal/equivalent to”
  - `!=` “is not equal/equivalent to”
  - `>` “is larger than”
  - `<` “is smaller than”
  - `>=` “is larger or equal to”
  - `<=` “is smaller or equal to”
  - `is` “is identical instance of”
  - `is not` “is not identical instance of”
  - `in` “is contained in collection”
  - `not in` “is not contained in collection”



**Loops**

**Functions**

**Classes,  
Modules &  
Packages**

**Programming  
Errors &  
Debugging**

# for-Loop

```
for «control variable name» in «iterable»:  
    «statement»
```

⚠ Mind the indentation!

# for-Loop: Iteration over ordered collections

## Loop over elements

```
1 # tuple filled with arbitrary elements  
2 my_tuple = (1, 2.0, 'text', list(), dict())  
3  
4 # for-loop over my_tuple with control  
   variable 'el'  
5 for el in my_tuple:  
6     msg = 'element: {}'.format(el)  
7     print(msg)
```

# for-Loop: Iteration over ordered collections

Loop over indices with `range`

```
1 # tuple filled with arbitrary elements
2 my_tuple = (1, 2.0, 'text', list(), dict())
3
4 # for-loop over my_tuple with control
   variable 'i'
5 for i in range(len(my_tuple)):
6     el = my_tuple[i]
7     msg = 'element {}: {}'.format(i, el)
8     print(msg)
```

# for-Loop: Iteration over ordered collections

Update `list` in for-loop

```
1 # list filled with arbitrary elements
2 my_list = [1, 2.0, 'text', list(), dict()]
3
4 # for-loop over my_list with control
   variable 'i'
5 for i in range(len(my_list)):
6     # update element with index i
7     my_list[i] = 'element {}: {}'.format(i,
8         my_list[i])
9     print(my_list[i])
```

# for-Loop: Iteration over ordered collections

Loop over indices and elements with `enumerate`

```
1 # list filled with arbitrary elements
2 my_list = [1, 2.0, 'text', list(), dict()]
3
4 # for-loop over my_list with control
  variables 'i' and 'el'
5 for i, el in enumerate(my_list):
6     # update element with index i
7     my_list[i] = 'element {}: {}'.format(i,
      el)
8     print('old: {}, new: {}'.format(el,
      my_list[i]))
```

# for-Loop: Iteration over unordered collections

Loop over elements of a **set**

```
1 # set filled with arbitrary elements  
2 my_set = {1, 1, 1, 2.0, 'text'}  
3  
4 # for-loop over my_set with control variable  
   'el'  
5 for el in my_set:  
6     msg = 'element: {}'.format(el)  
7     print(msg)
```

# for-Loop: Iteration over unordered collections

Loop over keys of a `dict`

```
1 # dictionary filled with arbitrary elements
2 my_dict = {'key': 'value', 1: 'text', (1, 2)
3           : 'text'}
4
5 # for-loop over keys of my_dict with control
6 variable 'key'
7 for key in my_dict:
8     val = my_dict[key]
9     msg = 'key: {}, value: {}'.format(key,
10    val)
11     print(msg)
```



# for-Loop: Iteration over unordered collections

Loop over items of a **dict**

```
1 # dictionary filled with arbitrary elements
2 my_dict = {'key': 'value', 1: 'text', (1, 2)
3           : 'text'}
4
5 # for-loop over items of my_dict with
6   control variables 'key', 'val'
7 for key, val in my_dict.items():
8     msg = 'key: {}, value: {}'.format(key,
9                                       val)
10    print(msg)
```

# Conditional iteration

Another type of loop in Python: `while`

- Loops until condition becomes True

```
1 x = 5
2 while x > 0:
3     print(x)
4     x -= 1 # shorthand for x = x - 1
```

## Special keywords in loops:

- `continue`: aborts current iteration and continues with the next
- `break`: aborts loop completely

# Quiz

- ❖ What does the instruction `tuple(range(3))` return?

[1, 2, 3]      (1, 2, 3)      (0, 1, 2)      (0, 1, 2, 3)

- ❖ Let  $x$  be any integer, how many times is the `print` statement in the following `for`-loop executed?

```
1 for i in range(x):  
2     for j in range(i):  
3         print((i, j))
```

# Quiz

- What does the instruction `tuple(range(3))` return?

[1, 2, 3]      (1, 2, 3)      (0, 1, 2) ✓      (0, 1, 2, 3)

- Let  $x$  be any integer, how many times is the `print` statement in the following `for`-loop executed?

```
1 for i in range(x):  
2     for j in range(i):  
3         print((i, j))
```

$\binom{x}{2}$  times

**Loops**

**Functions**

**Classes,  
Modules &  
Packages**

**Programming  
Errors &  
Debugging**

# Functions

```
def «functionName» ( «parameterName1»,  
«parameterName2», ... ):  
    «statement»  
    return «statement»
```

⚠ Mind the indentation!

*gray = optional*

# Variable Scope

## Functions have a separate variable scope!

- ❖ internal variables are not accessible from outside
- ❖ calling global functions and variables is possible
  - ❖ Reading global variables is discouraged
- ❖ Changing global variables requires

«`global` variableName»

statement inside function (highly discouraged)

source: <https://www.learnpython.org/en/Functions>

# Functions—a simple example

```
1 def myFirstFunction():  
2     print('this is my first function')  
3  
4 # call function  
5 myFirstFunction()  
6  
7 # save return value in variable  
8 hereComesNothing = myFirstFunction()
```



# Functions—example of code reuse

```
1 def findSubstringInStrings(stringCollection, pattern):
2     occ = list()
3     for i, s in enumerate(stringCollection):
4         j = s.find(pattern)
5         while j != -1:
6             occ.append((i, j))
7             j = s.find(pattern, j+1)
8     return occ
9
10 myStringList = ['the rain in spain', 'ain\t no sunshine',
11                'she was greeted with disdain']
12
13 occOfAin = findSubstringInStrings(myStringList, 'ain')
```

# Quiz

Have you ever seen a function calling itself? Consider the following:

```
1 def fun(x):  
2     if len(x) > 1:  
3         return fun(x[1:])  
4     return x
```

What does the function call `fun([1,2,3,4])` return?

# Quiz

Have you ever seen a function calling itself? Consider the following:

```
1 def fun(x):  
2     if len(x) > 1:  
3         return fun(x[1:])  
4     return x
```

What does the function call `fun([1,2,3,4])` return?

[4]

**Loops**

**Functions**

**Classes,  
Modules &  
Packages**

**Programming  
Errors &  
Debugging**

# Creating new types

- ❖ A `class` defines a new type
- ❖ It can provide
  - ❖ class variables & functions
  - ❖ instance variables & functions

# Classes—example of code reuse

```
1 class Library:
2     description = 'This is a Library'
3
4     def __init__(self, name):
5         # name the library
6         self.name = name
7         # create empty book storage on initialization
8         self.storage = list()
9
10    def addBook(self, book):
11        self.storage.append(book)
12
13    def getAllBooks(self):
14        return tuple(self.storage)
15
16 myLib = Library('Bodleian Library')
17 myLib.addBook('The Art of Computer Programming (D. Knuth)')
```

# Modules

- ❖ Every `.py` file is a module
- ❖ Modules can host functions, variables, and classes
- ❖ Imported modules with `import` statement
- ❖ Should not have blocks of code that are immediately executed
- ❖ Explicit reference to module scope: `global`
- ❖ Name of module available as global variable `__name__`

# Modules—example of code reuse

## mystringutils.py

```
1 #  
2 # A module for all kinds of string utils  
3 #  
4  
5 def findSubstringInStrings(stringCollection,  
6     pattern):  
7     occ = list()  
8     for i, s in enumerate(stringCollection):  
9         j = s.find(pattern)  
10        while j != -1:  
11            occ.append((i, j))  
12            j = s.find(pattern, j+1)  
13    return occ
```

## myscript.py

```
1 #!/usr/bin/env python3  
2  
3 import mystringutils  
4  
5 if __name__ == '__main__':  
6     myStringList = ['the rain in spain',  
7         'ain\t no sunshine',  
8         'she was greeted with disdain']  
9  
10    occOfAin = mystringutils.  
11        findSubstringInStrings(myStringList,  
12            'ain')  
13    print(occOfAin)
```



# Modules—example of code reuse

## mystringutils.py

```
1 #
2 # A module for all kinds of string utils
3 #
4
5 def findSubstringInStrings(stringCollection,
6                             pattern):
7     occ = list()
8     for i, s in enumerate(stringCollection):
9         j = s.find(pattern)
10        while j != -1:
11            occ.append((i, j))
12            j = s.find(pattern, j+1)
13    return occ
```

## myscript.py

```
1 #!/usr/bin/env python3
2
3 import mystringutils as su
4
5 if __name__ == '__main__':
6     myStringList = ['the rain in spain',
7                    'ain\'t no sunshine',
8                    'she was greeted with disdain']
9
10    occOfAin = su.findSubstringInStrings(
11        myStringList, 'ain')
12    print(occOfAin)
```

# Modules—example of code reuse

## mystringutils.py

```
1 #  
2 # A module for all kinds of string utils  
3 #  
4  
5 def findSubstringInStrings(stringCollection,  
6     pattern):  
7     occ = list()  
8     for i, s in enumerate(stringCollection):  
9         j = s.find(pattern)  
10        while j != -1:  
11            occ.append((i, j))  
12            j = s.find(pattern, j+1)  
13    return occ
```

## mymycript.py

```
1 #!/usr/bin/env python3  
2  
3 from mystringutils import  
4     findSubstringInStrings  
5  
6 if __name__ == '__main__':  
7     myStringList = ['the rain in spain',  
8         'ain\t no sunshine',  
9         'she was greeted with disdain']  
10  
11     occOfAin = findSubstringInStrings(  
12         myStringList, 'ain')  
13     print(occOfAin)
```

# Packages

- ❖ Way of structuring multiple modules into a directory hierarchy
- ❖ Package directories must contain a `__init__.py` file
- ❖ Can be imported the same way as modules
- ❖ Python itself offers many packages, and even more third-party packages are available through *package managers* such as `conda`

# Quiz

- ❖ In Python, a class is \_\_\_\_\_ for an object.
- a nuisance      an instance      a blueprint      a distraction

- ❖ Consider the following class:

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

What is the correct statement to instantiate a Dog object?

- ❖ Dog('Rufus', 3)
- ❖ Dog(self, 'Rufus', 3)
- ❖ Dog.\_\_init\_\_('Rufus', 3)

# Quiz

- ❖ In Python, a class is \_\_\_\_\_ for an object.
- a nuisance      an instance      a blueprint ✓      a distraction

- ❖ Consider the following class:

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

What is the correct statement to instantiate a Dog object?

- ❖ Dog('Rufus', 3) ✓
- ❖ Dog(self, 'Rufus', 3)
- ❖ Dog.\_\_init\_\_('Rufus', 3)

**Loops**

**Functions**

**Classes,  
Modules &  
Packages**

**Programming  
Errors &  
Debugging**

# Programming errors

Recognizing different types of errors:

- ❖ Syntactic: spelling & grammar mistakes
  - ❖ e.g.  $avg = (x y)/2$
- ❖ Semantic: mistakes in meaning, context, or program flow
  - ❖ e.g.  $avg = x + y/2$  or  $avg = (x + z)/0$

Distinction between

- ❖ Compile-time errors (syntactic, semantic)
- ❖ Runtime errors (semantic)

# RuntimeError

## Changing the size of my\_dict in loop

```
1 # dictionary filled with arbitrary elements
2 my_dict = {'key': 'value', 1: 'text', (1, 2)
3           : 'text'}
4
5 # for-loop over keys of my_dict with control
6   variable 'key'
7 for key in my_dict:
8     my_dict[(key, 1, 2, 3)] = 'new element'
```



# Catching exceptions

Controlled treatment of anticipated exceptions:

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

# Raising exceptions

Use **raise** keyword to throw exceptions:

```
1 def myFunction(collection):  
2  
3     if len(collection) == 0:  
4         raise RuntimeError("Invalid input: empty collection")  
5         # do something ..  
6         return  
7  
8 myFunction(list())
```

# Raising exceptions

Check properties of input parameters using the assert statement:

```
1 def myFunction(collection):  
2  
3     assert len(collection) > 0, "Invalid input: empty collection"  
4  
5     # do something ..  
6     return  
7  
8 myFunction(list())
```

Failed assertions result in an AssertionError

# Debugging

## PDB—the Python debugger

- ❖ Enables step-by-step proceeding of statements in Python programs
- ❖ Interaction with Python program at runtime
- ❖ Debugger is invoked by *breakpoints*
- ❖ Set breakpoint in arbitrary location of your code by
  - ❖ calling builtin “`breakpoint()`” function (Python version  $\geq 3.7$ )
  - ❖ statement “`import pdb; pdb.set_trace()`”

# Python debugger—example

```
1 # dictionary filled with arbitrary elements
2 my_dict = {'key': 'value', 1: 'text', (1, 2)
3           : 'text'}
4
5 # invoke Python debugger
6 breakpoint()
7
8 # for-loop over keys of my_dict with control
9 variable 'key'
10 for key in my_dict:
11     my_dict[(key, 1, 2, 3)] = 'new element'
```

# Quiz

- ❖ Is improper indentation a syntactic or semantic error?
- ❖ Consider the following code:

```
1 def str2int(x):  
2     try:  
3         return int(x)  
4         _____ ValueError:  
5         return -1
```

What keyword should be used here?

`except`

`raise`

`else`

`Exception`

source: <https://quizizz.com/>

# Quiz

- Is improper indentation a syntactic or semantic error? syntactic
- Consider the following code:

```
1 def str2int(x):  
2     try:  
3         return int(x)  
4         _____ ValueError:  
5         return -1
```

What keyword should be used here?

`except` ✓

`raise`

`else`

`Exception`

source: <https://quizizz.com/>

# Recap



# Summary

- ❖ `for` and `while`
- ❖ Code reuse through
  - ❖ Functions
  - ❖ Classes
  - ❖ Modules & Packages
- ❖ Compile-time and runtime errors
- ❖ Python debugger, a tool for hunting runtime errors (bugs)

# What comes next?

- ❖ Write your first function, class, module, and Python script
- ❖ Familiarize yourself with the Python Debugger
- ❖ Due date for this week's exercises is **Friday, May 8, 2020.**

*Next lecture:* Data management & analysis, Jupyter Notebook, text mining ... ..