# Programming

## Programming & Python Basics

```
"""
if extrapolate is None:
    extrapolate = self.extrapolate
x = np.asarray(x)
x_shape, x_ndim = x.shape, x.ndim
x = np.ascontiguousarray(x.ravel(), dtype=np
# With periodic extrapolation we map x to the
# [self.t[k], self.t[n]].
if extrapolate == 'periodic':
    n = self.t.size - self.k - 1
    x = self.t[self.k] + (x - self.t[self.k])

    extrapolate = False

out = np.empty((len(x), prod(self.c.shape[1:])),
self._ensure_c_contiguous()
self._evaluate(x, nu, extrapolate, out)
out = out.reshape(x_shape + self.c.shape[1:])
if self.axis != 0:
    # transpose to move the calculated values to t
    l = list(range(out.ndim))
    l = l[x_ndim:x_ndim+self.axis] + l[:x_ndim] + l
    out = out.transpose(l)
return out

def _evaluate(self, xp, nu, extrapolate, out):
    _bspl.evaluate_spline(self.t, self.c.reshape(self.c.s
            self.k, xp, nu, extrapolate, out)

def _ensure_c_contiguous(self):
    """
    c and t may be modified by the user. The Cython code e
    that they are C contiguous.
```

Luna Pianesi

Faculty of Technology, Bielefeld University

Bielefeld University, November 08, 2023

Loops

Functions

Classes, Modules & Packages

Programming Errors & Debugging

# Functions

```
def «functionName » ( «parameterName1»,
«parameterName2», ...  ):
␣␣␣␣«statement»                         ⚠ Mind the indentation!
␣␣␣␣return «statement»
```

*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*

```python
def myFirstFunction():
    print('this is my first function')

# call function
myFirstFunction()

# save return value in variable
hereComesNothing = myFirstFunction()
```

# Functions—example of code reuse

```python
def findSubstringInStrings(stringCollection, pattern):
    occ = list()
    for i, s in enumerate(stringCollection):
        j = s.find(pattern)
        while j != -1:
            occ.append((i, j))
            j = s.find(pattern, j+1)
    return occ

myStringList = ['the␣rain␣in␣spain', 'ain\'t␣no␣sunshine',
    'she␣was␣greeted␣with␣disdain']

occOfAin = findSubstringInStrings(myStringList, 'ain')
```

# Quiz

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

```python
def fun(x):
    if len(x) > 1:
        return fun(x[1:])
    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:

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

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

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

```
# dictionary filled with arbitrary elements
my_dict = {'key': 'value', 1: 'text', (1, 2): 'text'}

# for-loop over keys of my_dict with control variable 'key'
for key in my_dict:
    my_dict[(key, 1, 2, 3)] = 'new␣element'
```

# *Catching exceptions*

Controlled treatment of anticipated exceptions:

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

# *Raising exceptions*

Use `raise` keyword to throw exceptions:

```python
def myFunction(collection):

    if len(collection) == 0:
        raise RuntimeError("Invalid␣input:␣empty␣collection")
    # do something ..
    return

myFunction(list())
```

# *Raising exceptions*

Check properties of input parameters using the `assert` statement:

```python
def myFunction(collection):

    assert len(collection) > 0, "Invalid␣input:␣empty␣collection"

    # do something ..
    return

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

```python
# dictionary filled with arbitrary elements
my_dict = {'key': 'value', 1: 'text', (1, 2): 'text'}

# invoke Python debugger
breakpoint()

# for-loop over keys of my_dict with control variable 'key'
for key in my_dict:
    my_dict[(key, 1, 2, 3)] = 'new␣element'
```

# Quiz

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

```
def str2int(x):
    try:
        return int(x)
    ____________ ValueError:
        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:

```
def str2int(x):
    try:
        return int(x)
    __________ ValueError:
        return -1
```

What keyword should be used here?

except✓ raise else Exception

source: https://quizizz.com/

# Recap

# Summary

- Code reuse through functions
- 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 ***Wednesday, November 15, 2pm, 2023***.

*Next lecture:* Functional programming & lazy evaluation … …