Scientific Software Development with Python

Python Basics

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

Agenda

CHALMERS UNIVERSITY OF TECHNOLOGY

Learning outcomes

1. Moving beyond *just* programming:
   - Software planning and design
   - Collaboration with others
   - Ensuring correctness and reproducibility
   - Computing concepts
2. Software project as an opportunity to improve your research process

Disclaimer

- This is the first time I am giving a course, so there will be errors
- All course material is on github and you are very welcome to contribute anything from corrections to additional topics
- Software development is a craft and needs practice. Don’t expect too much from just listening in on the lectures.

Background

- interpreted, high-level and general-purpose programming language[1]
- Created by Guide von Rossum in the 90s
- Developed by Python Software Foundation

Advantages for scientific applications

- Free and open source implementations available
- Widely available
- Easy to debug
- Very readable code
- High productivity, extensive standard library (batteries included)

Disadvantages for scientific applications

- Slow for certain applications
- Weak typing can cause errors

Python Version

Don't use Python 2, it's dead (end of life was in January 2020).
Python 3 is all there is.

The IPython shell

- Start it using

```
$ ipython
```

  - Can be used to interactively execute Python code
  - Useful built-in functions: `type(...)`, `help`, `...?`

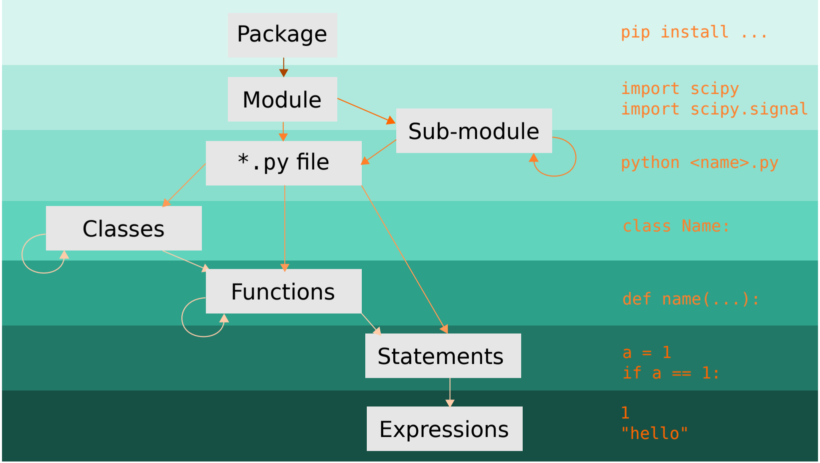
I encourage you to try and follow along with the lecture and execute some of the examples in IPython.

Editors and IDEs

- Advice: Take some time to setup your development environment, it's worth it.
- Features to look for:
  - Automatic PEP 8[2] formatting
  - Code completion
  - Syntax checking
  - Interactive shell
- Personal setup: Spacemacs with Python and syntax-checking layer.

[2] Python enhancement proposal (PEP) 8 is the official Python style guide.

CHALMERS
UNIVERSITY OF TECHNOLOGY


Package
Module
Sub-module
*.py file
Classes
Functions
Statements
Expressions
pip install ...
import scipy
import scipy.signal
python <name>.py
class Name:
def name(...):
a = 1
if a == 1:
1
"hello"

Package

- A collection of Python modules that can be distributed
- Can be made available online via a package index (pypi.org)
- Typical usage:

```
pip install <package_name>
```

CHALMERS
UNIVERSITY OF TECHNOLOGY

Modules

- Act as namespaces that bundle classes and functions
- Typical usage:

```
# Import statments tell Python to load a module
import module
import module as m
from module import function, Class

# Functions and classes defined in the module can
# be accessed through its attributes.
module.function()
m.function()
```

Functions

- Define a sequence of operations to be executed on a set of user-provided input variables.
- Example:

```python
def say_something(what):
    """ Prints given output to screen. """
    print(what)

say_something("hello") # Prints "hello"
```

Classes

- Define a set of properties (data) and associated behavior (functions)
- Methods are special functions that are associated with objects of the class.
- Example:

```
class A:
""" A class example. """
    def __init__(self):
    """ Create A object. """
        self.what = "hello"


    ...
```

- A statement is an instruction that can be interpreted by the Python interpreter.
- Examples:

```
# Import statement
import numpy as np

# Expression statement
f()

# Assignment statement
a = 1

...
```

Simple statements

- Simple statements are limited to a single line
- To extend a statement over multiple lines you can
  - Enclose the statement in parentheses (preferred)
  - Use line continuation: \

```
a = (1 +
     1)
b = 1 \
    + 1
```

CHALMERS
UNIVERSITY OF TECHNOLOGY

Compound statements

- Compound statements are statements that contain one or mutiple other statements
- The nested statements are grouped together by identical indentation (typically 4 spaces) and follow a colon:

```
if (True):
    print("Hi")
    print("there")

# Same as
if (True): print ("Hi\nthere")
```

- The following table gives an overview over keywords that introduce simple statements in Python[3]:

Keyword	Purpose
`assert`	Abort if condition is not met.
`pass`	NOOP, placeholder
`del`	Force deletion of object
`return`	In function: Return value from function
`raise`	Signals error
`break`	In loop: Abort loop
`continue`	In loop: Skip to next iteration
`global`	Use variable from global scope
`nonlocal`	Use variable in enclosing but non-global scope

[3] We'll see what most of them mean later on.

- The following table gives an overview over keywords that introduce compound statements in Python[3]:

Keyword	Purpose
`if`, `elif`, `else`	If statement
`while`	While loop
`for`	For loop
`try`, `except`, `finally`	Try, catch error, cleanup
`with`	Context manager
`def`	Function definition
`class`	Class definition
`async`	Coroutine

- Variables are defined through assignment statements:

```
a = 1
```

- Variables hold references to objects. This is important when working with *mutable* objects:

```
a = [1, 2]
b = a
b.append(3)
print(a) # Prints [1, 2, 3]
```

Think of Python variables as labels rather than containers.

- Valid variable names:
  - Begin with letter or underscore (_)
  - Followed by letter, number or underscore
- Variable names should be lowercase with words separated by underscore
- Exception: Constants should be all caps

```
SOME_CONSTANT = 42
some_variable = 1
```

- Objects are automatically deleted when they aren't reference by any variable (garbage collection) [4]

[4] This is why it is important to avoid cyclic references

- All Python object have a type that defines how they behave
- The type can be inferred using the built-in `type` function:

```
a_string = "hello"

# Prints: str
print(type(a_string))

# Prints documentation for variable type.
help(type(a_string))
```

Agenda

CHALMERS
UNIVERSITY OF TECHNOLOGY

Numeric literals

- Numeric literals are raw numbers that appear in Python code.

```
# Integral numbers
a = 0b10000 #binary literal
b = 0o20    #octal literal
c = 16      #decimal literal
d = 0x10    #hexadecimal literal
print (a == b == c == d) # Prints True

e = 1e6
f = 1_000_000
print (e == f) # Prints True

# Complex numbers
g = 1j
```

Operation	Operator	Example	Meaning
Addition	`+`	`a + b`	$a + b$
Subtraction	`-`	`a - b`	$a - b$
Multiplications	`*`	`a * b`	$a \cdot b$
Division (floating point)	`/`	`a / b`	$\frac{a}{b}$
Division (integer)	`//`	`a // b`	$\lfloor \frac{a}{b} \rfloor$
Modulus	`%`	`a % b`	$a - \lfloor \frac{a}{b} \rfloor \cdot b$
Exponent	`**`	`a ** b`	a^b

- All of these operators have compound versions which combine the operator with an assignment statement:

```
a += b # Same as a = a + b
a -= b # Same as a = a - b
...
```

Operation	Operator	Example
Logical and	`and`	`a == b`
Logical or	`or`	`a != b`
Logical not	`not`	`a > b`

- The two boolean literals are `True` and `False`.
- Logical operator have the lowest precedence of all operators. Parentheses are therefore usually not required but can make the code more readable.

```
a < b and b > c # Same as: (a < b) and (b > c)
```

Operation	Operator	Example
Equal	`==`	`a == b`
Not equal	`!+`	`a != b`
Greater than	`>`	`a > b`
Less than	`<`	`a < b`
Greater than or equal to	`>=`	`a >= b`
Less than or equal to	`<=`	`a <= b`

- Comparison operators can also be chained:

```
a == b == c   # Same as:(a == b) and (b == c)
a < b < c < d # Same as (a < b) and (b < c) ...
```

- Each separate object in a program has a unique identity
- The identity of two objects can be compared using the `is` and `is not` operators:

```
a = [] # Creates an empty list with name a
b = [] # Creates an empty list with name b
print(a is b) # Prints False
print(a == b) # Prints True
c = b
print(b is c) # Prints True
```

Use `is` only to check whether two variable point to *the same object* not when you want to compare two objects.

- String literals can be delimited using either single or double quotes:

```
a = "a 'string'"
b = 'another "string"'
```

- Multi-line strings are delimited using three ' or "

```
a = """a veeeeeeeeeeeeeeery
        veeeeeeeeeeeeeeery
        long
        string"""
```

- A wide range of common string operations is available via methods of the str class

Raw strings:

- A raw string is a string literal that is prefixed with `r`
- In normal strings, certain escape sequences starting with \ (backslash)

are replaced with special characters.

- In raw strings, this is not the case

```
print("\n") # Prints newline
print(r"\n") # Prints \n
```

- f-Strings ($\geq$ Python 3.6):

```python
answer = 42
text = f"The answer is {answer}."
print(text) # Prints: The answer is 42.
x = 1e-3
text = f"Advanced formatting: {x:07.4f}"
print(text) # Prints: Advanced formatting: 00.0010
```

- The `format` method:

```python
text = "The answer is {}.".format(42)
```

- See docs for full details on string methods.

```
# Lists are defined using brackets.
a_list = [1, 2, "three"]
empty_list = []

# Indexing is 0-based.
print(a_list[2]) # Prints: three

# Negative indices are counted backwards
# from the end
print(a_list[-1]) # Prints: three

# len returns length of the list
print(len(a_list)) # Prints 3
```

```python
# Reverse list
a_list = [1, 1, 2, 3]
a_list.reverse()
print(a_list) # Prints: [3, 2, 1, 1]

# Remove element from list
del a_list[0]
print(a_list) # Prints: [2, 1, 1]

# Remove first occurence from list
a_list.remove(1)
print(a_list) # Prints: [2, 1]

# Check presence of element in list
print(1 in a_list) # Prints: True
```

Slicing

- A slice is an expression of the form
  - `start:end`
  - or `start:end:step`
- Slicing always copies a list
- Slicing can be used to extract parts of lists:

```
a_list = [1, 2, 3, 4]

print(a_list[:])   # Prints: [1, 2, 3, 4]

print(a_list[2:])  # Prints: [3, 4]
print(a_list[1:3]) # Prints: [2, 3]
print(a_list[:2])  # Prints: [1, 2]

print(a_list[::2])  # Prints: [1, 3]
print(a_list[1::2]) # Prints: [2, 4]
print(a_list[-1:1:2]) # Prints: [4]
```

General form:

```
if condition:
    statement
```

- Can be followed by multiple `elif` and/or a single `else` block.
- Conditions are evaluated sequentially from left to right:

```
empty_list = []
# This raises no error although the list is empty
if (len(empty_list) > 0) and empty_list[0]:
    print(empty_list)
```

Instructions

- Go to `github.com/simonpf/bunny_lab`
- Scroll down
- Click on Colab badge

Time

- Time: 3 min + 2 min discussion in breakout rooms

General form:

```
for variable in iterable:
    statement
```

where `iterable` can be any object that *can be iterated over*[1].

Examples of iterables:

- lists
- tuples
- strings
- generators, e.g. `range(n)`

```python
# Prints 1, 2, 3, 4
for i in [1, 2, 3, 4]:
    print(i)

# Prints h e l l o
for c in "hello":
    print(i)

# Prints 0 h 1 e 2 l 3 l 4 o
for i, c in enumerate("hello"):
    print(i, c)

# Same as above.
for i, c in zip([1, 2, 3, 4], "hello"):
    print(i, c)
```

List comprehensions

- List comprehensions allow combining for-loops and if statements to generate a list:

```
numbers = [1, 2, 3, 4]
squares = [i ** 2 for i in numbers]
print(squares) # Prints: 1, 4, 9, 16

even_squares = [i ** 2 for i in numbers if i % 2 == 0]
print(even_squares) # Prints: 4, 16
```

- Like a list comprehension but enclosed with parentheses `(...)` instead of brackets `[...]`.
- Generators are lazy: Computation is deferred until elements are requested

```python
numbers = [1, 2, 3, 4]
# Prints: 1 2 3 4
say_numbers = [print(i) for i in numbers)]

# Prints nothing, yet.
say_numbers_lazy = (print(i) for i in numbers)
# Prints: 1 2 3 4
for i in say_numbers_lazy:
    pass
```

Exercise: Bunny Lab Level 2

CHALMERS
UNIVERSITY OF TECHNOLOGY

- Solve level 2 of the Bunny Lab
- Time: 3 min + 2 min discussion in breakout rooms

Tuples

- A sequence of objects separated by `,` and optionally enclosed with parentheses

```
t = (1, 2, 3)
```

- Tuples are immutable:

```
# Raises exception
t[0] = 2
```

- But mutable objects inside tuples can change:

```
t = ([], [], [])
t[0].append(1)
# Prints ([1], [], [])
print(t)
```

Converting between tuples and lists

- Use explicit constructors `list` and `tuple` to convert between them:

```
a_tuple = (1, 2, 3)
# Convert tuple to list
a_list = list(a_tuple)
# Append 4 to list
a_list += [4]
# Convert back to tuple
a_tuple = tuple(a_list)

# Prints (1, 2, 3, 4)
print(a_tuple)
```

Unpacking

- Python has special syntax for *unpacking* an iterable into multiple variables:

```
a, b, c = ["a", "b", "c"]
```

- These can be nested:

```
a, (b, c) = ["a", ["b", "c"]]
```

- Iterables can even be split up in specific elements and remaining sequences using *

```
first, body*, last = "a long string"

print(first)            # Prints: a
print("".join(body))    # Prints:  long strin
print(last)             # Prints: g
```

Overusing unpacking expressions can hurt readability.

- Don't overdo it: Limit unpacking to three variables[5]

[5] Slatkin, Brett. Effective Python

- Dictionaries can be used to store key-value pairs:

```
a_dictionary = {"key": "value"}
# Same as above
a_dictionary = dict([("key", "value")])
```

- Values can be retrieved:

```
# Prints: value
print(a_dictionary["key"])
```

- Values can be inserted into the dictionary:

```
a_dictionary["another_key"] = "another_value"
# Same as above
a_dictionary.update(("another_key", "another_value"))
```

- Looping over elements:

```
keys = [1, 2, 3, 4]
values = ["one", "two", "three", "four"]
dictionary = dict(zip(keys, values))

# Prints: 1 one 2 two ...
for key in dictionary:
    print(key, dictionary[key])

# Prints: 1 one 2 two ...
for key, value in dictionary.items():
    print(key, value)

# Prints: one two ...
for value in dictionary.value():
    print(value)
```

- Trying to retrieve a key that is not present in a dictionary raise an exception
- Can use `in` operator to check if key is present in dictionary
- Better approach: Use `get(...)` method to safely access elements in dict
  - Synopsis: `get(key, default=value)`
  - Returns `value` if key is not found in dictionary.

```
if key in dictionary:
    print(dictionary[key])
else:
    print("Key not found!")

# Same as above
print(dictionary.get(key, default="Key not found!"))
```

Exercise: Bunny Lab Level 3

- Solve level 3
- Time: 3 min + 2 min discussion in breakout rooms

1. Introduction
2. The Organisation of Python code
3. Syntax basics
4. Basic types and operators
5. **Functions**
6. Classes
7. Style and documentation
8. Summary

- Functions are defined using the `def` keyword follows:

```
def say_something(what):
    print(what)
```

- A function is called by its name followed by the required arguments in parentheses:

```
say_something("hello") # Prints "hello"
```

- A `return` statement in a function causes the execution to leave the function and return the specified value as result
- If no `return` statement is present, the function implicitly returns `None`

- Functions are also objects:

```
# Prints cryptic things
print(say_something)
# Prints the function name
print(say_something.__name__)
# Prints the function's doc string
print(say_something.__doc__)
```

- Functions can be passed as arguments to other functions:

```
def say_hello(): print("hello")

def do_something(what): what()

do_something(say_hello) # Prints "hello"
```

- Function parameters are passed by reference!

```python
def manipulate_list(input):
    input.reverse()

a_list = [1, 2, 3]
manupulate_list(a_list)

# Prints: [3, 2, 1]
print(a_list)
```

- The `lambda` keyword defines an anonymous function consisting of a single statement
- Synopsis:

```
lambda param_1, param_2, ...: statement
```

- The lambda function returns the evaluated `statement`

```
def do_something(what): what()

# Prints "hi"
do_something(lambda: print("hi"))
```

Python has two ways of passing arguments[6] to functions:

1. As positional arguments:

```python
def say_something(this, that):
    print(this)
    print(that)

# All function calls print: this that
say_something("this", "that")
```

[6]The terms parameters and arguments can be used interchangeably

Python has two ways of passing arguments to functions:

1. As keyword arguments:

```python
def say_something(this="this", that="that"):
    print(this)
    print(that)

# All functions print: this that
say_something()
say_something("this")
say_something("this", "that")
say_something(this="this")
say_something(that="that")
say_something(this="this", that="that")
```

Positional and keyword arguments

- Positional and keyword arguments can be mixed:

```
def say_something(what, this="this", that="that"):
    print(what)
    print(this)
    print(that)
```

- The caller *may* (but shouldn't) also give positional arguments in keyword form,

```
say_something(what="what")
```

- Positional arguments must always precede keyword arguments:

```
# This will raise an exception
say_something(this="this", that="that", "what")
```

- Variadic functions are functions that take a variable number of arguments
- Positional variadic arguments are declared using a starred expression[7]:

```
def say_something(*args): print(args)
# Prints: ["this", "that"]
say_something("this", "that")
```

- The arguments provided by the caller are available as `list` inside the function.

[7] Note similarity to parameter unpacking

- Keyword variadic arguments are declared using a double star expression:

```
def say_something(**kwargs): print(kwargs)
# Prints: {"this": "this", "that": "that"]
say_something(this="this", that="that")
```

- The arguments provided by the caller are available as `dict` inside the function.

Expanding parameter lists and dicts

- Note how the star and double-star expressions above pack the arguments provided by the caller into lists and dicts, respectively.
- It also works the other way around:

```
def say_something(this, that): print(this, that)

# Prints: "this" "that"
args = ["this", "that"]
say_something(*args)

# Prints: "this" "that"
kwargs = {"this": "this", "that": "that"}
say_something(**kwargs)
```

Exercise: Bunny Lab Level 4

- Time: 3 min + 2 min discussion in breakout rooms

CHALMERS
UNIVERSITY OF TECHNOLOGY

- Classes allow tying specific behavior to the data it depends on.
- Objects of a class are class are referred to as its instances
- The data associated with a class are called *attributes*
- The function associated with it are called *class methods*

```python
class Dog:
    def __init__(self, name):
        self.name = name

    def say_hi(self, who):
        print(f"Hi {who}, my name is {self.name}")

# Calls __init__ method.
dog = Dog("Charlie")

# Prints: Charlie
print(dog.name)

# Prints: Hi fren, my name is Charlie.
dog.say_hi("fren")
```

The `self` argument

- All class methods take `self` as first argument
- The `self` arguments refers to the class instance the method is called on.
- It is implicitly provided when an object's method is called using `.` notation
- It can also be provided explicitly when the method is called via the class name (second example)

```
dog.say_hi("fren")

# This is the same as
Dog.say_hi(dog, "fren")
```

Magic methods

CHALMERS
UNIVERSITY OF TECHNOLOGY

- *Magic* or *dunder* (from double underscore) methods are methods whose names begin and end with two underscores
- These functions often implement special functionality in Python
- Examples: `__init__`, `__add__`, `__getitem__`, ...

CHALMERS
UNIVERSITY OF TECHNOLOGY

```python
class Dog:
    def__init__(self, name):
        self.name = name

    def __add__(self, other):
        return Dog(self.name + " " + other.name)

dog_1 = Dog("Charlie")
dog_2 = Dog("Donut")
dog_3 = dog_1 + dog_2

# Prints: Hi fren, my name is Charlie Donut.
dog_3.say_hi("fren")
```

Exercise: Bunny Lab Level 5

- Time: 3 min + 2 min discussion in breakout rooms

Documentation

- Every module, class, (exported) function and (public) method should be described by a *docstring*.

```
class Dog:
   """
   The Dog class represents pet dogs.

   Attributes:
        name (str): The dog's name
   """
    def __init__(self, name):
        """
        Create dog instance.

        Args:
            name (str): The dog's name
        """
        self.name = name
```

- Note: Every object's docstring can be accessed from within Python via its `__doc__` attribute

Coding style

- PEP 8[1] provides a coding style guide for Python code:
  - Spaces instead of tabs
  - Line width: 79 characters
  - Imports a top of file
- Adhere to it, if you don't have a good reason not to.

Configure you editor or IDE to format your code according to PEP8. This way you can stop worrying about coding style and focus on actual programming.

Summary

- The concepts presented in this lecture should be enough to understand most Python programs
- However, there is much more: Keep an open mind, read other peoples code, you will learn new things all the time.
- Bunny lab:
    - Written using only standard library code, no external dependencies
    - Python code *can and should* tell a story