Programming Lab

Autumn Semester

Course code: PC503

Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 13

Modules

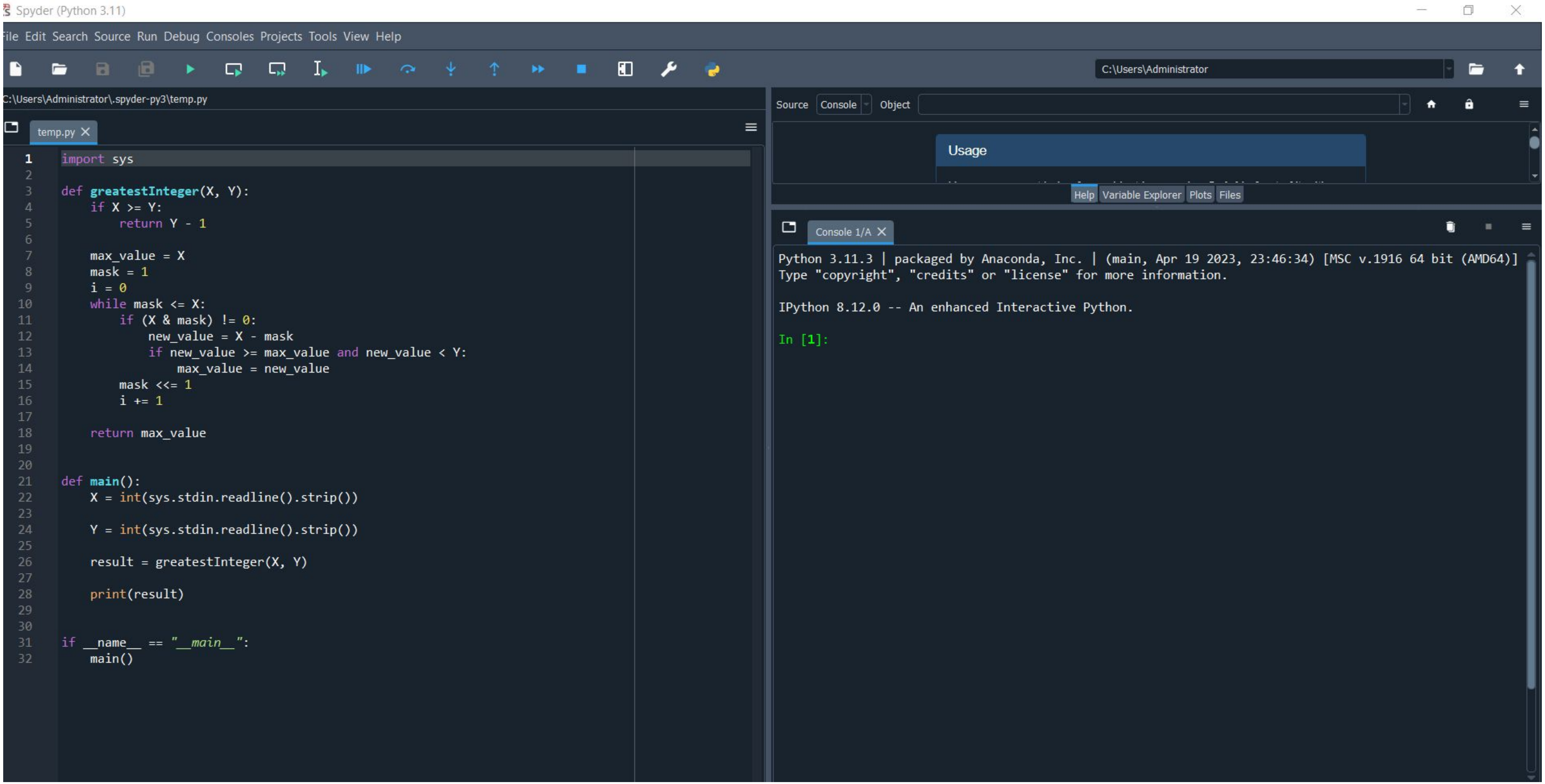
Comparing Sequences and Other Types

```
>>> (1, 2, 3)          < (1, 2, 4)
True
>>> (1, 2, 3)          < (1, 2, 4)
True
>>> [1, 2, 3]          < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> (1, 2, 3, 4)        < (1, 2, 4)
True
>>> (1, 2)              < (1, 2, -1)
True
>>> (1, 2, 3)           == (1.0, 2.0, 3.0)
True
>>> (1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Sequence objects typically may be compared to other objects with the same sequence type.

The comparison uses *lexicographical* ordering:

First, the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted.



Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Administrator
C:\Users\Administrator\.spyder-py3\temp.py
temp.py
import sys
def greatestInteger(X, Y):
    if X >= Y:
        return Y - 1
    max_value = X
    mask = 1
    i = 0
    while mask <= X:
        if (X & mask) != 0:
            new_value = X - mask
            if new_value >= max_value and new_value < Y:
                max_value = new_value
        mask <<= 1
        i += 1
    return max_value
def main():
    X = int(sys.stdin.readline().strip())
    Y = int(sys.stdin.readline().strip())
    result = greatestInteger(X, Y)
    print(result)
if __name__ == "__main__":
    main()
Source Console Object
Usage
Help Variable Explorer Plots Files
Console 1/A
Python 3.11.3 | packaged by Anaconda, Inc. | (main, Apr 19 2023, 23:46:34) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 8.12.0 -- An enhanced Interactive Python.
In [1]:

fibo.py

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

main.py

```
import fibo

fibo.fib(1000)

#if this not working
then use print()


fibo.fib2(100)

fibo.__name__
```

If you intend to use a function often you can assign it to a local name

main1.py

```
import fibo

fibo.fib(1000)

#if this not working
then use print()


fibo.fib2(100)

fibo.__name__


fib = fibo.fib
fib(500)
```

5 minutes assignment

Try for a factorial function or can make a calculator module with different operations?

More on Modules

- A module can ***contain executable statements as well as function definitions***.
- These statements are intended to initialize the module.
- They are executed only ***the first time the module name is encountered in an import statement***.
- Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.
- There is a variant of the import statement that imports names from a module directly into the importing module's namespace.

main2.py

```
from fibo import fib, fib2
fib(500)
#This does not introduce the module name from which the imports are taken in the local namespace
fibo.fib(5) # check out this….
```

- There is even a variant to import all names that a module defines:

main3.py

```
from fibo import *
fib(500)
```

- This imports all names except those beginning with an underscore (_).
- Note that in general the practice of importing * from a module or package is frowned upon since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

- If the module name is followed by as, then the name following as is bound directly to the imported module.

main4.py

```
import fibo as fib
fib.fib(500)
```

- This is effectively importing the module in the same way that import fibo will do, with the only difference of it being available as fib.

- It can also be used when utilizing from with similar effects:

main5.py

```
from fibo import fib as fibonacci
fibonacci(500)
```

Simple_math.py

```
# Importing built-in module math
import math

# using square root(sqrt) function contained
# in math module
print(math.sqrt(25))

# using pi function contained in math module
print(math.pi)

# 2 radians = 114.59 degrees
print(math.degrees(2))

# 60 degrees = 1.04 radians
print(math.radians(60))
```

Try to build your own module that performs similar operations geometry.py

```
# Sine of 2 radians
print(math.sin(2))

# Cosine of 0.5 radians
print(math.cos(0.5))

# Cosine of 0.5 radians
print(math.cos(0.5))

# Tangent of 0.23 radians
print(math.tan(0.23))

# 1 * 2 * 3 * 4 = 24
print(math.factorial(4))
```

Random_func.py

```
# importing built in module random
import random

# printing random integer between 0 and 5
print(random.randint(0, 5))

# print random floating point number between 0 and 1
print(random.random())

# random number between 0 and 100
print(random.random() * 100)

List = [1, 4, True, 800, "python", 27, "hello"]

# using choice function in random module for choosing
# a random element from a set such as a list
print(random.choice(List))
```

Date_and_time.py

```
# importing built in module datetime
import datetime
from datetime import date
import time

# Returns the number of seconds since the
# Unix Epoch, January 1st 1970
print(time.time())

# Converts a number of seconds to a date object
print(date.fromtimestamp(454554))
```

- For efficiency reasons, each module is only imported once per interpreter session.

- Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use importlib.reload(), e.g. import importlib; importlib.reload(modulename).

Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

The code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__". That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
   import sys
   fib(int(sys.argv[1]))
```

We can make the file usable as a script as well as an importable module because the code that parses the command line only runs if the module is executed as the "main" file:

```
python fibo.py 50
```

fibo1.py

```python
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Administrator: Command Prompt

```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>D:

D:\>cd D:\Programming course\IDE_Code

D:\Programming course\IDE_Code>python fibo1.py 100
0 1 1 2 3 5 8 13 21 34 55 89

D:\Programming course\IDE_Code>
```

Check if you do not provide any argument to the program:

```
C:\Users\Administrator>D:

D:\>cd D:\Programming course\IDE_Code

D:\Programming course\IDE_Code>python fibo1.py
```

If the module is imported, the code is not run:

```
D:\Programming course\IDE_Code>import fibo
'import' is not recognized as an internal or external command,
operable program or batch file.

D:\Programming course\IDE_Code>
```

The Module Search Path

- When a module named spam is imported, the interpreter first searches for a built-in module with that name.

- These module names are listed in sys.builtin_module_names.

- If not found, it then searches for a file named spam.py in a list of directories given by the variable sys. path. sys. The path is initialized from these locations:

  - *The directory containing the input script (or the current directory when no file is specified).*

  - *PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).*

  - *The installation-dependent default (by convention including a site-packages directory, handled by the site module).*

More details are at The initialization of the sys. path module search path.

“Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the __pycache__ directory under the name **module.version.pyc**, where the version encodes the format of the compiled file; it generally contains the Python version number.

For example, in CPython release 3.3 the compiled version of spam.py would be cached as **__pycache__/spam.cpython-33.pyc**. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it’s out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. *First, it always recompiles and does not store the result for the module that’s loaded directly from the command line.* Second, *it does not check the cache if there is no source module.*

Standard Modules

- Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter).
- Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls.
- The set of such modules is a configuration option that also depends on the underlying platform. For example, the **winreg** module is only provided on Windows systems.
- One particular module deserves some attention: sys, which is built into every Python interpreter.
- The variables **sys.ps1** and **sys.ps2** define the strings used as primary and secondary prompts:

Standard Modules

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> sys.ps2 = 'rahul'
C> if 1==0:
rahul
```

- The variable **sys.path** is a list of strings that determines the interpreter’s search path for modules.

- It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations:

```
C> import sys
C> sys.path.append('/ufs/guido/lib/python')
C>
```

The dir() Function

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
import fibo, sys
dir(fibo)
Out[5]: ['__builtins__',
 '__cached__',
'__doc__',
'__file__',
'__loader__',
 '__name__',
  '__package__',
 '__spec__',
 'fib',
 'fib2']
```

dir(sys)

['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__', '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__', '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework', '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook', 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix', 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing', callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth', 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix', 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info', 'warnoptions']

- Without arguments, dir() lists the names you have defined currently:

```
a = [1, 2, 3, 4, 5]
import fibo
fib = fibo.fib
dir()
```

- Note that it lists all types of names: variables, modules, functions, etc.

- dir() does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module builtins:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'ModuleNotFoundError',.............................'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
>>>
```

Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names".
- For example, the module name A.B designates a submodule named B in a package named A.
- Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.
- Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data.
- There are many different sound file formats (usually recognized by their extension, for example .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats.
- There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, and creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations.
- Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```