計算物理概論

Introduction to Computational Physics (PHYS290000)

Lecture 5: Basic Python (part 3)

Instructor: 潘國全

kuochuan.pan@gapp.nthu.edu.tw

Last week

1. Warm-up
2. Exercise: Angry bird game
3. Formatted output
4. Read/Write files
5. Exercise: Angry bird game

Today’s plan

1. Exercise: Angry bird game (continue)
2. Modules / Packages
3. Classes (part 1)
4. Exercise: Angry bird game (version 2)

The Angry bird game (continue)

Angry bird

So far, we have these features in our angry bird game

1. Takes arbitrary inputs (separated by spaces)
2. Could quit the game by typing “quit”, “Quit”, or “QUIT”
3. Has the ability to check if hitting the target
4. Record the trajectory
5. Game continues for unlimited turns

Angry bird

(See demo)

Angry bird

- Additional features can be added (such as GUI interface, history, multiple targets, multiple game stages, scores, …etc.).
- However, if we write all these features in a single python file, the code will become difficult to read and to maintain.
- Could we separate different features into different python codes? Yes, that is modules.

Modules

Modules

1. A module is a file consisting of python codes
2. A module can define functions, classes and variables
3. A module allows you to logically organize your python codes
4. Grouping related code into a module makes the code easier to read and use
5. Modules can be used by the “import” statement
6. Recall the “import math”

Examples

File: particle.py

ComputationalPhysics > tutorial > particle.py > ...

```python
def set_particle_mass():
    print("set particle mass.")
    return

def apply_forces():
    print("force applied.")
    return

```

File: physics.py

ComputationalPhysics > tutorial > physics.py > ...

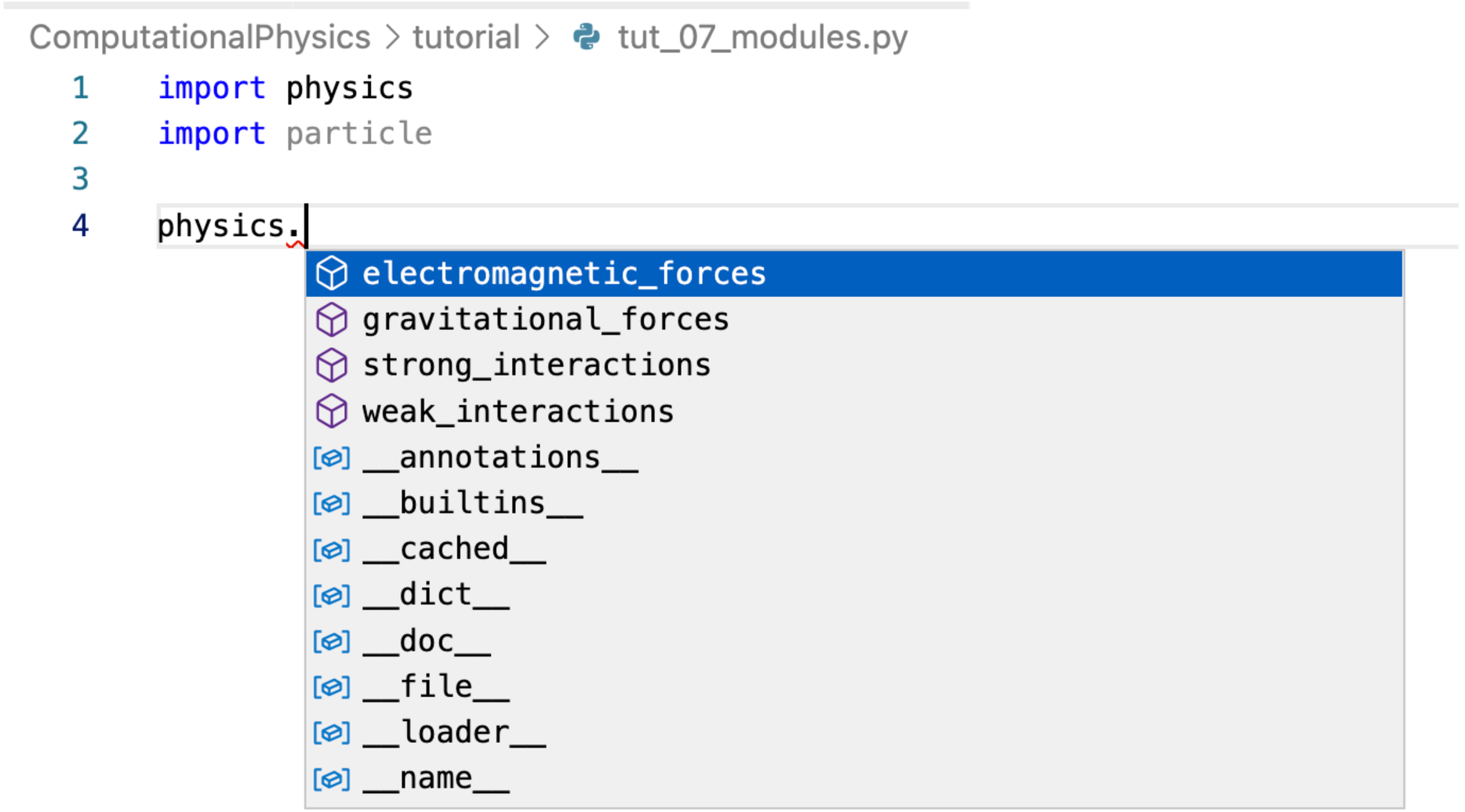
```python
def weak_interactions():
    print("weak forces.")
    return

def strong_interactions():
    print("strong forces.")
    return

def gravitational_forces():
    print("gravitational forces.")
    return

def electromagnetic_forces():
    print("EM forces.")
    return
```

Example


ComputationalPhysics > tutorial > tut_07_modules.py
1 import physics
2 import particle
3
4 physics.
electromagnetic_forces
gravitational_forces
strong_interactions
weak_interactions
__annotations__
__builtins__
__cached__
__dict__
__doc__
__file__
__loader__
__name__

Example

ComputationalPhysics > tutorial > tut_07_modules.py > ...

```python
import physics
import particle

def engine():
    physics.weak_interactions()
    physics.gravitational_forces()
    particle.apply_forces()

if __name__=='__main__':

    engine()

```

```
----------
● ~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_07_modules.py
weak forces.
gravitational forces.
force applied.
```

Import

1. import physics
2. import physics as phy
3. from physics import *
4. from physics import weak_interactions
5. from physics import weak_interactions as weak

Exercise: Angry bird game, again!

1. Now, let’s separate the code of our angry bird game into several modules.
2. Modules include: game.py, physics, and output.py. Which function should put into which file?

Standard modules

1. There are many standard python library: https://docs.python.org/3/library/
2. The system module (import sys). For example, print(sys.argv[1]; sys.path.append(‘’); sys.path.insert(1, “”)).
3. Import math
4. Datetime: import datetime
5. Additional non-standard libraries can be installed via “conda install” or “pip install”
6. Try “conda install numpy”

Packages

Python packages

1. Packages are a way of structuring Python's module namespace by using "dotted module names".

2. For example, the module name A.B designates a submodule named B in a package named A.

```
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
              ...
```

https://docs.python.org/3/tutorial/modules.html

Python packages

1. Packages are a way of structuring Python's module namespace by using "dotted module names".
2. For example, the module name A.B designates a submodule named B in a package named A.
3. The __init__.py files are required to make Python treat directories containing the file as packages.

Python packages

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)
```

Intra-package references

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

Classes (part 1)

Python Class

1. Classes provide a means of bundling data and functionality together.
2. Classes are a fundamental concept in object-oriented programming (OOP)
3. Classes allow you to create your own data types with their own attributes and methods.
4. Classes are the foundation of many Python libraries and frameworks, such as matplotlib, spicy, …etc.

Creating a class

1. To create a class, use the “class” keyword followed by the class name.

2. Class names should start with a capital letter.

ComputationalPhysics > tutorial > tut_08_class.py > ...

```
1    class PhysicalConstant:
2        pass
3
```

Defining Attributes

1. Attributes are variables defined within a class

2. They are also known as instance variables or properties

```python
class PhysicalConstant:
    """
    A class to include physical constants
    """
    def __init__(self, units="cgs"):
        """
        :param units: supprot "cgs" and "mks".
        """
        self.units = units
        self.c_cgs = 2.99792458e10
        self.c_mks = 2.99792458e8
        self.G_cgs = 6.67428e-8
        self.G_mks = 6.67428e-11
        if units=="cgs":
            self.c = self.c_cgs
            self.G = self.G_cgs
        elif units=="mks":
            self.c = self.c_mks
            self.G = self.G_mks
        else:
            print(r"Eror: no such units. units = {units}")
            print(r"Set the units to 'cgs'. ")
            self.c = self.c_cgs
            self.G = self.G_cgs

        return
```

Defining methods

1. Methods are functions defined inside a class that can be called on an object of the class. To define a method, you create a function inside the class. The first argument of the method should always be self, which refers to the object calling the method.

```
28      def set_units(self, units):
29          self.units = units
30          if units=="cgs":
31              self.c = self.c_cgs
32              self.G = self.G_cgs
33          elif units=="mks":
34              self.c = self.c_mks
35              self.G = self.G_mks
36          else:
37              print(r"Eror: no such units. units = {units}")
38              print(r"Set the units to 'cgs'. ")
39              self.c = self.c_cgs
40              self.G = self.G_cgs
41          return
```

Creating objects

1. To create an object of a class, you call the class name as if it were a function and pass in any arguments required by the __init__ method.

```
43  if __name__=='__main__':
44
45      const = PhysicalConstant(units="cgs")
46      print(f"[{const.units}] c = {const.c:12.6e}")
47      const.set_units("mks")
48      print(f"[{const.units}] c = {const.c:12.6e}")
49
```

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_08_class.py
[cgs] c = 2.997925e+10
[mks] c = 2.997925e+08
```

Example: The Angry bird game

1. Back to the angry bird game.

2. Let’s use classes to re-write our angry bird game.

(See demo)