計算物理概論

Introduction to Computational Physics (PHYS290000)

Lecture 7: Advanced Python (part 1)

Instructor: 潘國全

kuochuan.pan@gapp.nthu.edu.tw

Last week

1. Python Classes
2. Property, getter, setter
3. Data class
4. Exercise: Angry bird game (again!)
5. Homework 2 is released

Today’s plan

1. Advanced topics
2. Errors and Exceptions
3. Command line arguments
4. Numerical integral
5. Lambda
6. Performance measurement

Errors and Exceptions

Errors and Exceptions

1. In the angry bird, we already learned that we could use “try” and “except” to handle some error messages, but actually these error messages have two distinguishable kinds of errors: errors and exceptions.
2. Syntax Errors / IndentationError (parsing errors): missing “:” missing “tabs”

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python
Python 3.10.9 | packaged by conda-forge | (main, Feb  2 2023, 20:24:27) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def print_hello()
  File "<stdin>", line 1
    def print_hello()
                    ^
SyntaxError: expected ':'
>>>
```

Built-in Exceptions

1. See: https://docs.python.org/3/library/exceptions.html#bltin-exceptions
2. Common exceptions

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Ref: https://www.devinline.com/2015/04/exception-handling-in-python.html

Handling Exceptions

1. There are possibility that you only want to handle exceptions of certain errors (not all errors)

```python
def input_velocity():
    try:
        x = float(input("Enter a real number.\n"))

    except ValueError:

        print("Error: please enter a real number. Try again.")

    return
```

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_11_errors.py
Enter a real number.
pan
Error: please enter a real number. Try again.
```

Handling Exceptions

2. You can handle multiple errors at once

```
try:
    f = open("trajectory.txt",mode="r")
    s = f.readline()
    print(s.strip())
    i = int(s.strip())
except OSError as err:
    print("OS Error:", err)
except ValueError:
    print("Could not convert data to an integer")
except Exception as err:
    print(f"Unexpected error {err=},{type(err)=}")
    raise
else:
    print(s)
    f.close()
```

Raising errors

1. Use “raise” to raise an error

```
try:
    if float(velocity) >= 2.997924581e8:
        raise RuntimeError(f"{velocity=} can not be faster than c.")
    print("The velocity looks good")
except ValueError as exec:
    raise RuntimeError("Incorrect value of velocity.") from exec
```

User defined exceptions

1. To make the code easier to read, we could define our custom error exceptions.

```
class MyError(Exception):
    pass

class InvalidVelocityError(Exception):
    def __init__(self):
        msg = "Velocity cannot be faster than the speed of the light."
        super().__init__(msg)
```

Extra useful tips

1. Use “finally” to clean-up actions.
2. Add notes in the exception (after python 3.11)

```
try:
    raise TypeError('bad type')
except Exception as e:
    e.add_note("We can add extra notes.") # after python 3.11
    e.add_note("more info about this error.")
    raise
```

Exercise: Error and exception

1. Write a function to set the temperature of a fiber in a lab. The Lab requires a low temperature environment with T < 30 K.

```
def set_temperature(newT):
    # do stuff here
    return newT
```

2. Raise two types of errors when (1) UnphysicalTemperatureError: when the desired temperature is less or equal to 0 K (2) HighTemperatureError: when temperature is higher than 30K
3. Make an exception with the HighTemperatureError, but print a warning to the user and store the new temperature in a log file (“temperature.log”).

Command Line Arguments

Command line arguments

1. There are many situations that you want reuse the same codes but with different parameters.
2. You could either modify the constant every time your run the code, or use the sys.argv to set the parameter (like the angry bird game we used)
3. A more professional way is to use the “argparse” package.

Argparse

```
from argparse import ArgumentParser

def set_parser():

    parser = ArgumentParser()
    parser.add_argument('fname')
    parser.add_argument('-v','--velocity',type=float,default=100,help="the velocity [m/s]")
    parser.add_argument('-u','--units',type=str, default="mks", help="The units systems. 'cgs' or 'mks'")
    parser.add_argument('-d','--debug',dest="is_debugging",action="store_true",default=False,help="use debug mode")
    return parser

def get_parameters(parser):

    args = parser.parse_args()
    print(f"{args.fname=}")
    print(f"{args.velocity}")
    print(f"{args.units}")
    print(f"{args.is_debugging}")
    return
```

```
if __name__=='__main__':

    parser = set_parser()
    get_parameters(parser)
```

Argparse

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_12_argparse.py
usage: tut_12_argparse.py [-h] [-v VELOCITY] [-u UNITS] [-d] fname
tut_12_argparse.py: error: the following arguments are required: fname
```

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_12_argparse.py --help
usage: tut_12_argparse.py [-h] [-v VELOCITY] [-u UNITS] [-d] fname

positional arguments:
  fname

options:
  -h, --help            show this help message and exit
  -v VELOCITY, --velocity VELOCITY
                        the velocity [m/s]
  -u UNITS, --units UNITS
                        The units systems. 'cgs' or 'mks'
  -d, --debug           use debug mode
```

Argparse

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_12_argparse.py trajectory.txt
args.fname='trajectory.txt'
100
mks
False
```

```
~/codes/ComputationalPhysics/ComputationalPhysics/tutorial (main*) » python tut_12_argparse.py trajectory.txt -v 30.5 --debug
args.fname='trajectory.txt'
30.5
mks
True
```

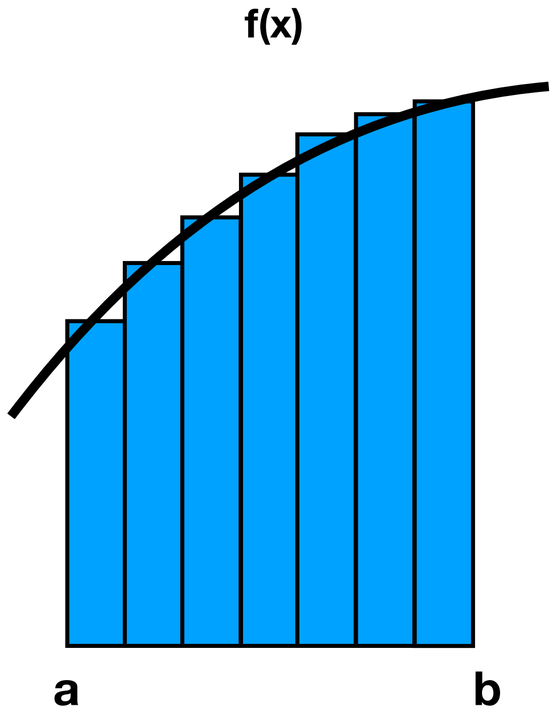
Exercise: The Angry bird game (again)

1. Modify the “angry.py” file in the google classroom (you could disable the plot_trajectory() function if you have problem to use matplotlib).
2. Modify the play() function. Replace the sys.argv parts to argparse.
3. Implement options with
   - - - velocity (-v) to set the initial velocity (default = 20 m/sec)
   - - - angle (-a) to set the initial inclination angle (default= 40 degree)
   - - - plot to plot the trajectory (default: False)
   - - - log to write into a log file (default: True)

Example: Numerical Integral

Write a numerical integrator

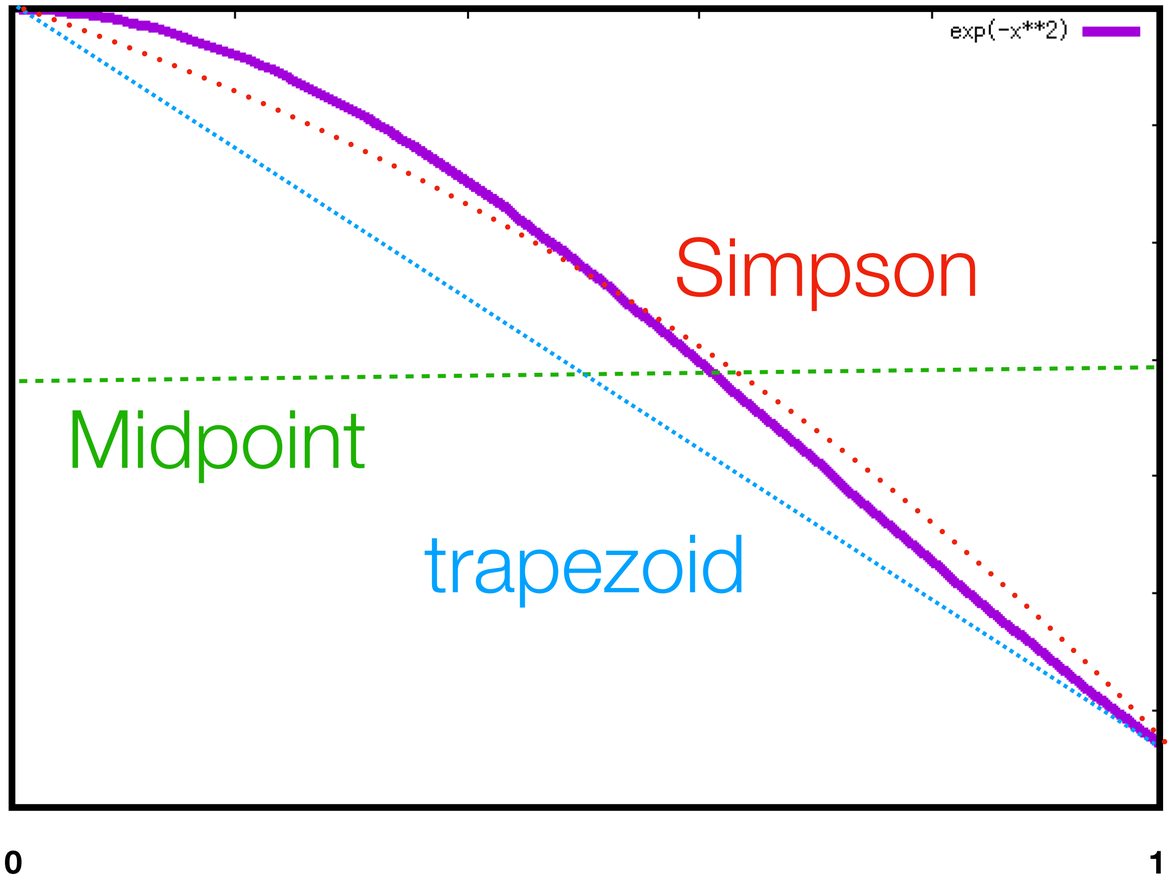
1. To integrate a function from a to b is to evaluate the area under the function in [a,b]



2. Let's write a python class called "Integrator" that can do finite integral of an arbitrary function from a to b with dx = [b-a]/N, where N is the user-defined, number of divisions.

3. To evaluate the area of each sub-devision, we could use "midpot", "trapezoid", or "simpson" methods

Write a numerical integrator



4.

Midpoint rule

$$\int_a^b f(x)dx \sim (b-a)f(\frac{a+b}{2})$$

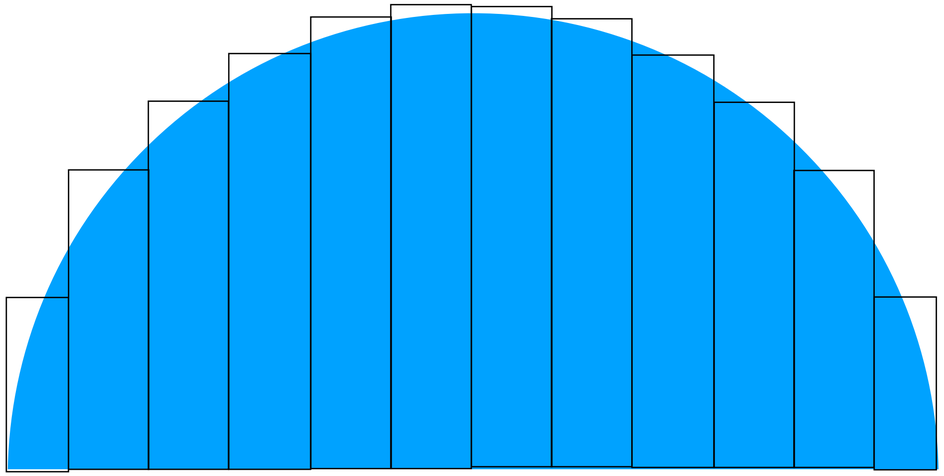
Trapezoidal rule

$$\int_a^b f(x)dx \sim (b-a)(\frac{f(a)+f(b)}{2})$$

Simpson's rule

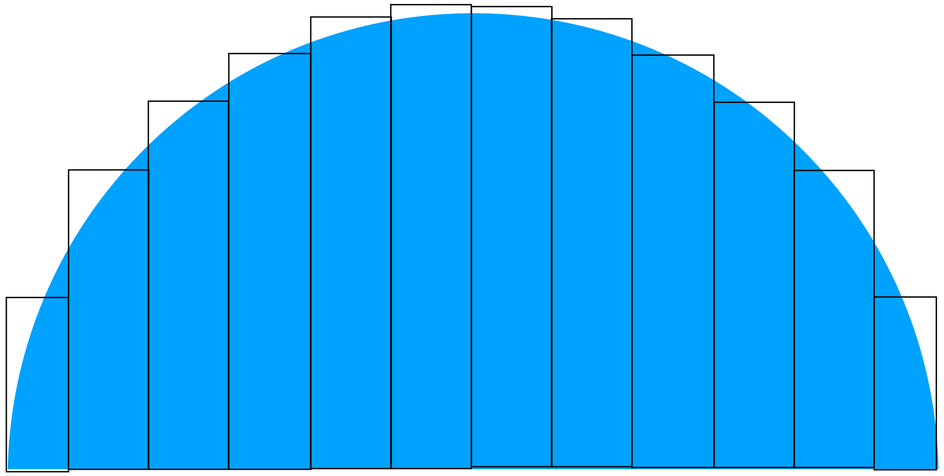
$$\int_a^b f(x)dx \sim \frac{(b-a)}{6}(f(a)+4f(\frac{a+b}{2})+f(b))$$

Example: Calculate Pi



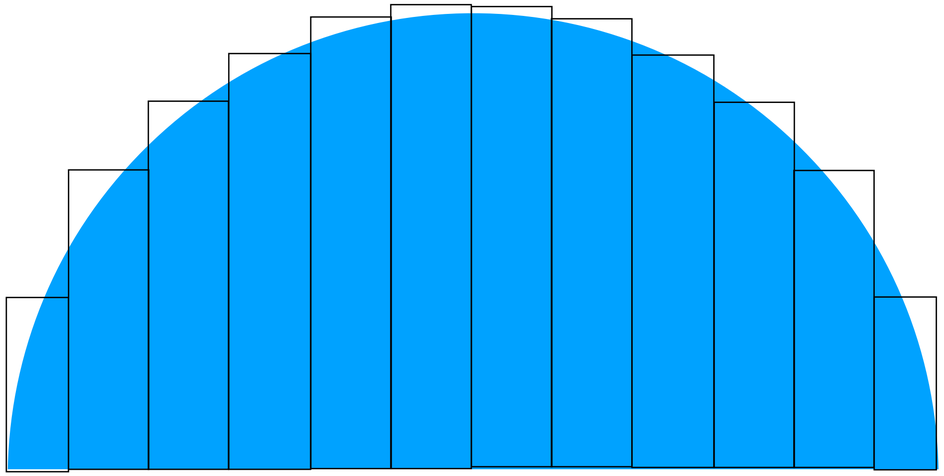
1. The area of a half-unit-circle is equal to pi/2
2. Use the integrator we wrote to evaluate pi
3. Start from the “midpoint” method.
4. See the sample code “integrator_template.py”
5. Note: the program could be slow in this example

Example: Calculate Pi



```python
import math

class Integrator:

    def __init__(self, func):
        """
        setup the integrator with a given function
        """

        # TODO

        return

    def midpoint(self,a=-1,b=1,N=10_000):
        """
        Assume a < b
        """

        # TODO

        return area


if __name__=='__main__':

    def hcirc(x):
        return math.sqrt(1-x**2)

    integrate = Integrator(func=hcirc)
    area = integrate.midpoint(a=-1,b=1,N=10000)
    print(2*area)
```

Exercise: other methods



1. Now, it is your time to practice implementing the trapezoidal rule and Simpson's rule.
2. Write two class methods "trap" and "simpson" to do the same integrals.
3. Try different N, see your results with different methods are less or larger than the true PI.
4. Remember to raise an error if b <= a

Python Lambda

Write a numerical integrator

1. Python's "lambda" is a very powerful way to define functions.
2. A lambda function is a small anonymous function (no function name)
3. A lambda function can take any number of arguments, but can only have one expression
4. Syntax: `lambda arguments : expression`
5. Example

```
hcirc = lambda x: math.sqrt(1-x**2)
```

Write a numerical integrator

6. More examples:

```
# declare a lambda function
greet = lambda : print("Hello world!")

# call the lambda function
greet()
```

7. Create a list in one line with lambda

```
list1 = [(lambda x: x**2)(x) for x in range(10)]
#or
list2 = [x**2 for x in range(10)]
print(list1)
print(list2)
```

Write a numerical integrator

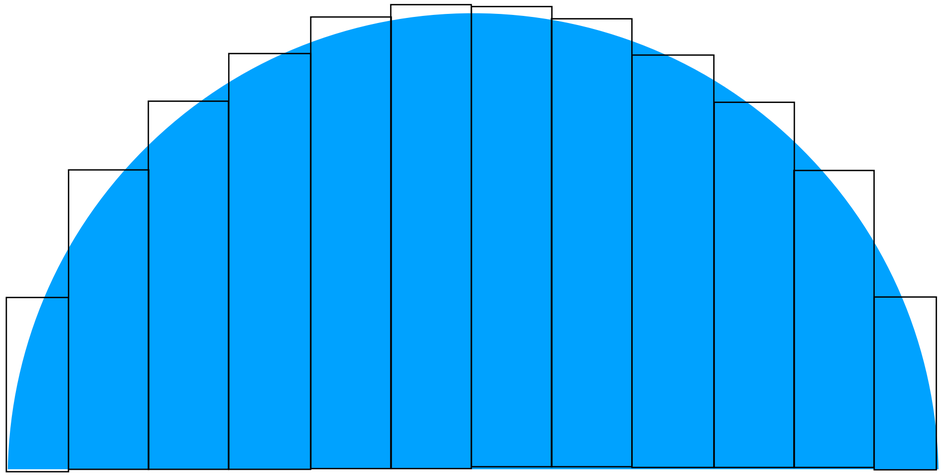
8. Use lambda to map a list

```
a  = [1, 2, 3, 4, 5, 6]
b = list(map(lambda x: x * 2 , a))

#or
c = [x*2 for x in a]
print(b)
print(c)
```

9. Use lambda to filter a list

```
a = [1, 2, 3, 4, 5, 6, 7, 8]
b = list(filter(lambda x: (x%2 == 0) , a))

# or
c = [x for x in a if x%2 == 0]
print(b)
print(c)
```

Exercise 2: use lambda



1. Back to your Integrator(), now your lambda to define your function
2. Write a new method called midpoint2. Use lambda or list comprehensions to avoid the for loop when summing the area
3. Measure the performance of the two methods (midpoint and midpoint2)
4. (Next week we will show that both methods are very slow)

Performance Measurement

Timer and performance measurement

1. Performance measurement is very import in scientific computing.
2. Python could be VERY SLOW if we code it improperly.
3. It is very easy to make > 100 times slower than other programming language with python
4. When every you finish a program, you should think where is the performance bottleneck and whether we could improve it or not.

Timer and performance measurement

1. A simple way to measure the performance

```python
import time

def func1():
    # do some calculations
    a = 1
    for n in range(1000):
        a += 1
    print(a)
    return

if __name__=='__main__':

    t1 = time.time()
    func1()
    t2 = time.time()
    print(f"Time spend = {t2-t1}")
```

Timer and performance measurement

2. Use the “timeit” module

```
import timeit
timer = timeit.Timer(func1)
t = timer.timeit(number=10_000)
print(f"Average time = {t/10_000} sec.")
```

Timer and performance measurement

2. Use the “timeit” module to measure the performance of a function with arguments

```
def func2(N=1_000):
    # do some calculations
    a = 1
    for n in range(N):
        a += 1
    return
```

```
timer = timeit.Timer(lambda: func2(N=1000))
t = timer.timeit(number=1_000)
print(f"Average time = {t/1_000} sec.")
```

Exercise: Create a list

1. Measure the performance of creating a list with N=1_000_000 ones (i.e. [1,1,1….,1]).

2. Use

```python
def make_list1(N=1_000_000):
    a = []
    for n in range(N):
        a.append(1)
```

3. Use

```python
def make_list2(N=1_000_000):
    a = [1 for n in range(N)]
    return a
```

4. Remember that the CPU clock speed is about ~GHz (or ~ 1ns per operation). We should expect that it takes only 1e6 (elements) x 1 ns ~ 1.e-6 sec to create the list. What are the values you observed? Is python fast?

5. (Demo): use numpy

Exercise: Looping a list

1. Measure the performance of summing all the N=1_000_000 elements.
2. Use

```
def loop1(a):
    total = 0
    for v in a:
        total += v
    return total
```

3. Use the builtin “sum” function (i.e. total = sum(a))
4. (Demo): use numpy
5. Conclusions: Python’s default “loop” is VERY SLOW. In physical problem, we often encounter situation with big “array” or “matrix” (lists). We should avoid using loops. Scientific packages like “numpy” and “scipy” are therefore recommended.