SEE1002
Introduction to Computing for Energy and Environment

Part 3: Basic Python programming

Sec. 3: Good programming practices

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Fundamentals

Section 2: Branching or Decision Making

Section 3: Loops

Section 4: Functions

Part 3: Basic Python programming

Section 1: Modules

Section 2: Structure of a Python program

Section 3: Good programming practices

Section 4: Derived data structures

Part 4: Python for science and engineering

Section 1: Vectors, matrices and arrays

Section 2: Useful mathematical and statistical functions

Section 3: Plotting

Section 4: Input and output

Part 5: Applications

Motivation

Review

We've already discussed the elements of a good computer program.

1. Does what it's supposed to do (*correctness*)
2. Doesn't waste time doing unnecessary things *(efficiency)*
3. Isn't longer than necessary *(conciseness)*
4. Is easy to follow *(readability)*

Approaches

We've already discussed structured programming.

- It **directly addresses** conciseness and readability.
- It **indirectly addresses** correctness.

Structured programming is a popular way of improving the quality of a program, but it's not the only way…

Structured programming versus object-oriented programming

Structured programming is a so-called programming model or paradigm. It emphasizes the division of a program into blocks and sub-blocks (i.e. functions).

Object-oriented programming emphasizes the creation of data structures or objects. Most modern computer languages (including Python) support objects. **We will not cover objects in this course** (it's covered in second-year CS courses).

Nevertheless, it's useful to know a little bit about objects.

A simple example

Imagine that the GPA of BEng, MSc and PhD students is calculated using different formulas.

- **Structured programming.** The grades of each student are stored in a list, e.g. `grades`; the GPA is calculated by calling the appropriate function, e.g. `gpa_beng(grades)` or `gpa_msc(grades)`.
- **Object-oriented programming.** Student information (including programme and grades) is stored in an object, `student`. The gpa is calculated by calling a function or method associated with the object, e.g. `student.gpa()` that takes the student's registration into account. The referencing is similar to that for modules.

Specific techniques

We want to review some simple techniques for improving programs.

1. Comments *(readability)*
2. Naming of variables *(readability)*
3. Exception handling *(correctness)*
4. Appropriate data structures *(conciseness)*

N.B. we're not going to address efficiency.

1. Comments

Review

Comments are **text descriptions that are ignored by Python.** Comments follow #. They can be used anywhere within a line.

```
# this entire line is a comment
```

```
statement # comment follows statement
```

Why are comments useful?

Comments improve readability for the original programmer and anybody who has to use the program.

1. Background info on program (e.g. purpose, history)
2. Label blocks and sub-blocks
3. Explain method
4. Working notes
5. Debugging

For programmer

Example 1 (Sec 3.2, Example 1 revisited)

```
# Example 1: commented version of Sec. 3.1, example 1a (multiple functions

#------------------------
# functions
# ----------------------
def areaRectangle(L,W):
    area=L*W
    return (area)

def perimeterareaRectangle(L,W):
    return (L*W,2*(L+W))

def perimeterRectangle(L,W):
    return (2*L + 2*W)

#------------------------
# main program
# ----------------------

# 1. Input
L=float(input('Enter length: '))
W=float(input('Enter width: '))

# 2. Call functions and output
print( 'The area of the rectangle=',areaRectangle(L,W) )
print( 'The perimeter of the rectangle=',perimeterRectangle(L,W) )

area,perimeter=perimeterareaRectangle(L,W) # two return values
print( 'The area of the rectangle=',area )
print( 'The perimeter of the rectangle=',perimeter )
```

background

main sections

explanatory text

Example 1b (Lab 2.4e, Exercise 3)

```
# Example 1b: commented version of Lab 2.4e, Exercise 3
# Calculate the factorial using a for loop.
# Since n! = nx n-x x ... 1, we can start with fact=1 and loop to n.
# The value of the factorial is updated during each pass through the loop.

# get n
n = int(input('Enter a positive integer: '))

# calculate factorial
i = 1   # counter variable
fact = 1 # current value of factorial

while i <= n: # keep looping until we reach n
    fact = fact*i # update factorial
    i = i+1

print(fact)
```

explain method

explain variables

explain loop

Example 1c (Sec. 3.2, Example 2b revisited)

```
from math import *

def areaRectangle(L,W):
    '''
    calculate area of rectangle \n
    input: L (length), W (width) \n
    output: area
    '''
    area=L*W
    return (area)

def areaCircle(r):
    '''
    calculate area of circle
    input: r (radius)
    output: area
    '''
    area=pi*r**2
    print( 'L=', L )  # can we access L?
    return (area)

def main():
    L = 1.0 # L and w are local variables
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )


main()
```

We can leave working notes for ourselves

Example 1d (Sec. 3.2, Example 2b revisited)

```
from math import *

def areaRectangle(L,W):
    '''
    calculate area of rectangle \n
    input: L (length), W (width) \n
    output: area
    '''
    area=L*W
    return (area)

def areaCircle(r):
    '''
    calculate area of circle
    input: r (radius)
    output: area
    '''
    area=pi*r**2
    # test access to global variables
    # print( 'L=', L )
    # print( 'W=', W )
    return (area)

def main():
    L = 1.0 # L and w are local variables
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )


main()
```

We can comment out code to disable it. We can uncomment it if we want to use it later.

How detailed should comments be?

- For this course, you should add enough so that your code is clear to yourself and the TAs.
- For large programming projects, some additional background may be included (e.g. references on methods for solving equations).
- Generally comments should be used sparingly. Too many comments makes code harder to read.

2. Naming

Appropriate names

Choosing appropriate names sounds trivial but it can make things a lot easier.

- Names should be informative/suggestive (e.g. `area`, `L`, `W`).
- Names shouldn't be too long
- Instead of spaces one can use the underscore (e.g. `initial_guess`) or CamelCase (e.g. `InitialGuess`). The latter form is more popular in Python.

Variable conventions

There are certain conventions that are usually followed in scientific programing:

- Use `i,j,k,l` for integer variables (i.e. loops) and nest them in this order.
- Use `N` or `M,N` for number of points.
- Use `dt,dx`, etc. for timestep or gridspacing.

Choosing sensible names makes things easier!

3. Exception handling

What is exception handling?

Exception handling refers to the treatment of special cases or errors.

Exceptions can be raised via:

- Unexpected input from keyboard (or file)
- Illegal operations

Strictly speaking, exception handling is optional. But it **increases the robustness of programs**. It's more or less essential for programs that will be used by more than one user.

Example 2: input exception

```
r=float(input('Enter radius: '))
area=pi*r*r
print(area)
```

r is supposed to be a number

```
Enter radius: hello
Traceback (most recent call last):
```

```
    r=float(input('Enter radius: '))

ValueError: could not convert string to float: 'hello'
```

text input raises an exception

Example 3: calculation exception

```
a=1.0
b=0.0

print(a/b)
```

divide by zero

```
    print(a/b)

ZeroDivisionError: float division by zero
```

This is a very common error!

i) Exception avoidance

Exceptions can sometimes be avoided with an `if` test. The idea is that we don't carry out an operation that generates an exception.

```
a=1.0
b=0.0

if b == 0:
    print( 'error! b must be non-zero' )
else:
    print( a/b )
```

```
error! b must be non-zero
```

Limitations

Manual exception avoidance only works if we can anticipate the exception, i.e., we need to add the `if` test ahead of time.

It also requires us to consider each case individually.

ii) Explicit exception handling

Manual exception avoidance doesn't work for cases in which the exception can't be anticipated. It also requires us to deal with each case separately.

The goal of exception handling is to deal with all exceptions gracefully.

```
r=float(input('Enter radius: '))
area=pi*r*r
print(area)
```

number is expected

There's no way to avoid generating an exception if the user enters text.

`try-except`

`try-except` allow us to deal with exceptions in Python.

```
try:
    statement1
    [...]
except:
    statement2
    [...]
```

try executing this block

execute this code block if there's an exception

Example 2b: input using `try-except`

```
from math import pi

try:
    r = float(input('Enter radius: '))
    area = pi*r*r
    print( area )
except:
    print( 'r must be a number' )
```

try to calculate area

error message

```
Enter radius: hello
r must be a number
```

```
Enter radius: 1.0
3.141592653589793
```

Comments

1. One can have multiple else blocks corresponding to specific exceptions (e.g. `ZeroDivisionError`). We will not cover this.

2. One can also have an `else` block that's executed only if there are no exceptions.

When should exception handling be used?

- Exception handling should be added whenever there's a section of code that could cause problems.
- With practice, such sections can be identified by sight. Alternatively exception handling can be added afterwards.
- `try-except` yields neater code and is more general. However, it isn't always necessary.

Summary

1. Comments are useful in several ways but they should be used judiciously.
2. Undesirable behaviour (i.e. runtime errors) can be avoided with exception avoidance or exception handling.