24 Miscellany

"Efficient is better..."

Contents

KanNotes

The topics discussed in this chapter are far too removed from the mainstream Python programming for inclusion in the earlier chapters. These topics provide certain useful programming features, and could prove to be of immense help in certain programming strategies.

Documentation Strings

- It is a good idea to mention a documentation string (often called doscstring) below a module, function, class or method definition. It should be the first line below the **def** or the **class** statement.
- The docstring is available in the attribute **__doc__** of a module, function, class or method.
- If the docstring is multi-line it should contain a summary line followed by a blank line, followed by a detailed comment.
- Single-line and Multi-line docstrings are written within triple quotes.
- Using **help( )** method we can print the functions/class/method documentation systematically.
- In the program given below the function **display( )** displays a message and the function **show(msg1, msg2)** displays **msg1** in lowercase and **msg2** in uppercase. It uses a single line docstring for **display( )** and a mulit-line docstring for **show( )**. It displays both the docstrings. Also, it generates help on both the functions.

```
def display( ) :
    """Display a message"""
    print('Hello')
    print(display.__doc__)

def show(msg1 = ' ', msg2 = ' ') :
    """Display 2 messages

    Arguments:
    msg1 -- message to be displayed in lowercase (default ' ')
    msg2 -- message to be displayed in uppercase (default ' ')
    """
    print(msg1.lower( ))
    print(msg2.upper( ))
    print(show.__doc__)

display( )
show('Cindrella', 'Mozerella')
help(display)
help(show)
```

On execution of the program it produces the following output:

```
Hello
Display a message.
cindrella
MOZERELLA
Display 2 messages.

        Arguments:
        msg1 -- message to be displayed in lowercase (default ' ')
        msg2 -- message to be displayed in uppercase (default ' ')

Help on function display in module __main__:

display( )
    Display a message.

Help on function show in module __main__:

show(msg1=' ', msg2=' ')
    Display 2 messages.

    Arguments:
    msg1 -- message to be displayed in lowercase (default ' ')
    msg1 -- message to be displayed in uppercase (default ' ')
```

Command-line Arguments

- Arguments passed to a Python script are available in **sys.argv**.

```
# sample.py
import sys
print('Number of arguments received = ', len(sys.argv))
print('Arguments received = ', str(sys.argv))
```

If we execute the script as

```
C:\>sample.py cat dog parrot
```

we get the following output:

```
Number of arguments received = 4
Arguments received = sample.py  cat  dog  parrot
```

- If we are to write a program for copying contents of one file to another, we can receive source and target filenames through command-line arguments.

```
# filecopy.py
import sys
import shutil
argc = len(sys.argv)
if argc != 3 :
    print('Incorrect usage')
    print('Correct usage: filecopy source target')
else :
    source = sys.argv[1]
    target = sys.argv[2]
    shutil.copyfile(source, target)
```

Parsing of Command-line

- While using the 'filecopy.py' program discussed above, the first filename is always treated as source and second as target. Instead of this, if we wish to have flexibility in supplying source and target filenames, we can use options at command-line as shown below:

  ```
  filecopy.py  -s  phone  -t newphone
  filecopy -t newphone -s phone
  filecopy -h
  ```

  Now argument that follows **-s** would be treated as source filename and the one that follows **-t** would be treated as target filename. The option **-h** is for receiving help about the program.

- To permit this flexibility, we should use the **getopt** module to parse the command-line.

```
# filecopy.py
import sys, getopt
```

```
import shutil
if len(sys.argv) == 1 :
    print('Incorrect usage')
    print('Correct usage: filecopy.py -s <source> -t <target>')
    sys.exit(1)

source = ''
target = ''
try :
    options, arguments = getopt.getopt(sys.argv[1:],'hs:t:')
except getopt.GetoptError :
    print('filecopy.py -s <source> -t <target>')
else :
    for opt, arg in options :
        if opt == '-h' :
            print('filecopy.py -s <source> -t <target>')
            sys.exit(2)
        elif opt == '-s' :
            source = arg
        elif opt == '-t' :
            target = arg
    else :
        print('source file: ', source)
        print('target file: ', target)
        if source and target :
            shutil.copyfile(source, target)
```

- **sys.argv[1:]** returns the command-line except the name of the program, i.e. **filecopy.py.**
- Command line and the valid options are passed to **getopt( )**. In our case the valid options are **-s**, **-t** and **-h**. If an option has an argument it is indicated using the **:** after the argument, as in **s:** and **t:**. **-h** option has no argument.
- The **getopt( )** method parses **sys.argv[1:]** and returns two lists—a list of (option, argument) pairs and a list of non-option arguments.
- Some examples of contents of these two lists are given below:

  Example 1:

  ```
  filecopy.py  -s  phone  -t newphone
  ```

options would be [('-s', 'phone'), ('-t', 'newphone')]
arguments would be [ ]

Example 2:

filecopy.py -h

options would be [('-h', ' ')]
arguments would be [ ]

Example 3:

filecopy.py -s phone -t newphone word1 word2

options would be [('-s', 'phone'), ('-t', 'newphone')]
arguments would be ['word1', 'word2']

- Note that non-option arguments like **word1**, **word2** must always follow option arguments like **-s**, **-t**, **-h**, otherwise they too would be treated as non-option arguments.
- **sys.exit( )** terminates the execution of the program.
- IDLE has no GUI-based provision to provide command-line arguments. So at command prompt you have to execute program as follows:

  C:\>idle.py -r filecopy.py -s phone -t newphone

  Here **-r** indicates that when IDLE is launched it should run the script following **-r**.
- When we are experimenting with **getopt( )** function, frequently going to command-prompt to execute the script becomes tedious. Instead you can set up **sys.argv[ ]** at the beginning of the program as shown below:

  sys.argv = ['filecopy.py', '-s', 'phone', '-t', 'newphone']

Bitwise Operators

- Bitwise operators permit us to work with individual bits of a byte. There are many bitwise operators available:

  ~ - not (also called complement operator)
  << - left shift, >> - right shift
  & - and, | - or, ^ - xor

- Bitwise operators usage:

```
ch = 32
dh = ~ch              # toggles 0s to1s and 1s to 0s
eh = ch << 3          # << shifts bits in ch 3 positions to left
fh = ch >> 2          # >> shifts bits in ch 2 positions to right
a = 45 & 32           # and bits of 45 and 32
b = 45 | 32           # or bits of 45 and 32
c = 45 ^ 32           # xor bits of 45 and 32
```

- Remember:

  Anything ANDed with 0 is 0.
  Anything ORed with 1 is 1.
  1 XORed with 1 is 0.
  << - As bits are shifted from left, zeros are pushed from right.
  >> - As bits are shifted from right, left-most bit is copied from left.

- Purpose of each bitwise operator is given below:

~	- Convert 0 to 1 and 1 to 0
<< >>	- Shift out desired number of bits from left or right
&	- Check whether a bit is on / off. Put off a particular bit
|	- Put on a particular bit
^	- Toggle a bit

- Bitwise in-place operators: <<= >>= &= |= ^=

  a = a << 5 is same as **a <<= 5**
  b = b & 2 is same as **b &= 2**

- Except ~ all other bitwise operators are binary operators.

Assertion

- An assertion allows you to express programmatically your assumption about the data at a particular point in execution.

- Assertions perform **run-time checks** of assumptions that you would have otherwise put in code comments.

```
# denominator should be non-zero, i.e. numlist must not be empty
avg = sum(numlist) / len(numlist)
```

Instead of this, a safer way to code will be:

```
assert len(numlist) != 0
avg = sum(numlist) / len(numlist)
```

If the condition following **assert** is true, program proceeds to next instruction. If it turns out to be false then an **AssertionError** exception occurs.

- Assertion may also be followed by a relevant message, which will be displayed if the condition fails.

```
assert len(numlist) != 0, 'Check denominator, it appears to be 0'
avg = sum(numlist) / len(numlist)
```

- Benefits of Assertions:
  - Over a period of time comments may get out-of-date. Same will not be the case with assert, because if they do, then they will fail for legitimate cases, and you will be forced to update them.
  - Assert statements are very useful while debugging a program as it halts the program at the point where an error occurs. This makes sense as there is no point in continuing the execution if the assumption is no longer true.
  - With assert statements, failures appear earlier and closer to the locations of the errors, which makes it easier to diagnose and fix them.

Decorators

- Functions are 'first-class citizens' of Python. This means like integers, strings, lists, modules, etc. functions too can be created and destroyed dynamically, passed to other functions and returned as values.
- First class citizenship feature is used in developing decorators.
- A decorator function receives a function, adds some functionality (decoration) to it and returns it.
- There are many decorators available in the library. These include the decorator **@abstractmethod** that we used in Chapter 20.
- Other commonly used library decorators are **@classmethod**, **@staticmethod** and **@property**. **@classmethod** and **@staticmethod**

decorators are used to define methods inside a class namespace that are not connected to a particular instance of that class. The **@property** decorator is used to customize getters and setters for class attributes.

- We can also create user-defined decorators, as shown in the following program:

```
def my_decorator(func) :
    def wrapper( ) :
        print('****************')
        func( )
        print('~~~~~~~~~~~~~~~~')
    return wrapper

def display( ) :
    print('I stand decorated')

def show( ) :
    print('Nothing great. Me too!')

display = my_decorator(display)
display( )
show = my_decorator(show)
show( )
```

On executing the program, we get the following output.

```
****************
I stand decorated
~~~~~~~~~~~~~~~~
****************
Nothing great. Me too!
~~~~~~~~~~~~~~~~
```

- Here **display( )** and **show( )** are normal functions. Both these functions have been decorated by a decorator function called **my_decorator( )**. The decorator function has an inner function called **wrapper( )**.

- Name of a function merely contains address of the function object. Hence, in the statement

display = my_decorator(display)

we are passing address of function **display( )** to **my_decorator( )**. **my_decorator( )** collects it in **func**, and returns address of the inner function **wrapper( )**. We are collecting this address back in **display**.

- When we call **display( )**, in reality **wrapper( )** gets called. Since it is an inner function, it has access to variable **func** of the outer function. It uses the address stored in **func** to call the function **display( )**. It prints a pattern before and after this call.
- Once a decorator has been created, it can be applied to multiple functions. In addition to **display( )**, we have also applied it to **show( )** function.
- The syntax of decorating **display( )** is complex for two reasons. Firstly, we have to use the word display thrice. Secondly, the decoration gets a bit hidden away below the definition of the function.
- To solve both the problems, Python permits usage of @ symbol to decorate a function as shown below:

```
def my_decorator(func) :
    def wrapper( ) :
        print('*****************')
        func( )
        print('~~~~~~~~~~~~~~~~~')
    return wrapper

@my_decorator
def display( ) :
    print('I stand decorated')

@my_decorator
def show( ) :
    print('Nothing great. Me too!')

display( )
show( )
```

Decorating Functions with Arguments

- Suppose we wish to define a decorator that can report time required for executing any function. We want a common decorator which will work for any function regardless of number and type of arguments that it receives and returns.

```
import time

def timer(func) :
    def calculate(*args, **kwargs) :
        start_time = time.perf_counter( )
        value = func(*args, **kwargs)
        end_time = time.perf_counter( )
        runtime = end_time - start_time
        print(f'Finished {func.__name__!r} in {runtime:.8f} secs')
        return value
    return calculate

@timer
def product(num) :
    fact = 1
    for i in range(num) :
        fact = fact * i + 1
    return fact

@timer
def product_and_sum(num) :
    p = 1
    for i in range(num) :
        p = p * i + 1

    s = 0
    for i in range(num) :
        s = s + i + 1

    return (p, s)

@timer
def time_pass(num) :
    for i in range(num) :
        i += 1

p = product(10)
print('product of first 10 numbers =', p)
p = product(20)
print('product of first 20 numbers =', p)
fs = product_and_sum(10)
```

```
print('product and sum of first 10 numbers =', fs)
fs = product_and_sum(20)
print('product and sum of first 20 numbers =', fs)
time_pass(20)
```

Here is the output of the program...

```
Finished 'product' in 0.00000770 secs
product of first 10 numbers = 986410
Finished 'product' in 0.00001240 secs
product of first 20 numbers = 330665665962404000
Finished 'product_and_sum' in 0.00001583 secs
product and sum of first 10 numbers = (986410, 55)
Finished 'product_and_sum' in 0.00001968 secs
product and sum of first 20 numbers = (330665665962404000, 210)
Finished 'time_pass' in 0.00000813 secs
```

- We have determined execution time of three functions—**product( )**, **product_and_sum( )** and **time_pass( )**. Each varies in arguments and return type. We are still able to apply the same decorator **@timer** to all of them.
- The arguments passed while calling the three functions are received in ***args** and ****kwargs**. This takes care of any number of positional arguments and any number of keyword arguments that are needed by the function. They are then passed to the suitable functions through the call

  value = func(*args, **kwargs)
- The value(s) returned by the function being called is/are collected in **value** and returned.
- Rather than finding the difference between the start and end time of a function in terms of seconds a performance counter is used.
- **time.perf_counter( )** returns the value of a performance counter, i.e. a clock in fractional seconds. Difference between two consecutive calls to this function determines the time required for executing a function.
- On similar lines it is possible to define decorators for methods in a class.

Unicode

- Unicode is a standard for representation, encoding, and handling of text expressed in all scripts of the world.
- It is a myth that every character in Unicode is 2 bytes long. Unicode has already gone beyond 65536 characters—the maximum number of characters that can be represented using 2 bytes.
- In Unicode every character is assigned an integer value called code point, which is usually expressed in Hexadecimal.
- Code points for A, B, C, D, E are 0041, 0042, 0043, 0044, 0045. Code points for characters अ आ इ ई उ of Devanagari script are 0905, 0906, 0907, 0908, 0909.
- Computers understand only bytes. So we need a way to represent Unicode code points as bytes in order to store or transmit them. Unicode standard defines a number of ways to represent code points as bytes. These are called encodings.
- There are different encoding schemes like UTF-8, UTF-16, ASCII, 8859-1, Windows 1252, etc. UTF-8 is perhaps the most popular encoding scheme.
- The same Unicode code point will be interpreted differently by different encoding schemes.
- Code point 0041 maps to byte value 41 in UTF-8, whereas it maps to byte values ff fe 00 in UTF-16. Similarly, code point 0905 maps to byte values e0 a4 85 and ff fe 05 \t in UTF-8 and UTF-16 repsectively. You may refer table available at https://en.wikipedia.org/wiki/UTF-8 (https://er for one to one mapping of code points to byte values.
- UTF-8 uses a variable number of bytes for each code point. Higher the code point value, more the bytes it needs in UTF-8.

bytes Datatype

- In Python text is always represented as Unicode characters and is represented by **str** type, whereas, binary data is represented by **bytes** type. You can create a **bytes** literal with a prefix **b**.

```
s = 'Hi'
print(type(s))
print(type('Hello'))
by = b'\xe0\xa4\x85'
print(type(by))
print(type(b'\xee\x84\x65'))
```

will output

```
<class 'str'>
<class 'str'>
<class 'bytes'>
<class 'bytes'>
```

- We can't mix **str** and **bytes** in concatenation, in checking whether one is embedded inside another, or while passing one to a function that expects the other.
- Strings can be encoded to bytes, and bytes can be decoded back to strings as shown below:

```
eng = 'A B C D'
dev = 'अ आ  इ ई'

print(type(eng))
print(type(dev))
print(eng)
print(dev)

print (eng.encode('utf-8') )
print (eng.encode('utf-16') )
print (dev.encode('utf-8') )
print (dev.encode('utf-16') )

print(b'A B C D'.decode('utf-8'))
print(b'\xff\xfeA\x00 \x00B\x00 \x00C\x00 \x00D\x00'
    .decode('utf-16'))
print(b'\xe0\xa4\x85 \xe0\xa4\x86 \xe0\xa4\x87\xe0\xa4\x88'
    .decode('utf-8'))
print(b'\xff\xfe\x05\t \x00\x06\t \x00\x07\t \x00\x08\t'
    .decode('utf-16'))
```

Execution of this program produces the following output:

```
<class 'str'>
```

```
<class 'str'>
अ आ इ ई
A B C D
b'A B C D'
b'\xff\xfeA\x00 \x00B\x00 \x00C\x00 \x00D\x00'
b'\xe0\xa4\x85 \xe0\xa4\x86 \xe0\xa4\x87 \xe0\xa4\x88'
b'\xff\xfe\x05\t \x00\x06\t \x00\x07\t \x00\x08\t'
A B C D
A B C D
अ आ इ ई
अ आ इ ई
```

- How these Unicode code points will be interpreted by your machine or your software depends upon the encoding scheme used. If we do not specify the encoding scheme, then the default encoding scheme set on your machine will be used.

- We can find out the default encoding scheme by printing the value present in **sys.stdin.encoding**. On my machine it is set to UTF-8.

- So when we print **eng** or **dev** strings, the code points present in the strings are mapped to UTF-8 byte values and characters corresponding to these byte values are printed.

Create Executable File

- If we are developing a program for a client, rather than giving the source code of our program, we would prefer to given an executable version of it. The steps involved in creating the executable file are given below:

- Step 1: Install the Pyinstaller Package

  In the Windows Command Prompt, type the following command to install the pyinstaller package (and then press Enter):

  ```
  C:\Users\Kanetkar>pip install pyinstaller
  ```

- Step 2: Go to folder where the Python script is stored.

  ```
  C:\Users\Kanetkar>CD Programs
  ```

- Step 3: Create the Executable using Pyinstaller

  ```
  C:\Users\Kanetkar\Programs>pyinstaller --onefile ScriptName.py
  ```

- Step 4: Executable file pythonScriptName.exe will be created in 'dist' folder. Double-click the EXE file to execute it.

P</> Programs

Problem 24.1

Write a program that displays all files in current directory. It can receive options -h or -l or -w from command-line. If -h is received display help about the program. If -l is received, display files one line at a time,. If -w is received, display files separated by tab character.

Program

```
# mydir.py
import os, sys, getopt

if len(sys.argv) == 1 :
    print(os.listdir('.'))
    sys.exit(1)

try :
    options, arguments = getopt.getopt(sys.argv[1:],'hlw')
    print(options)
    print(arguments)
    for opt, arg in options :
        print(opt)
        if opt == '-h':
            print('mydir.py -h -l -w')
            sys.exit(2)
        elif opt == '-l' :
            lst = os.listdir('.')
            print(*lst, sep = '\n')
        elif opt == '-w' :
            lst = os.listdir('.')
            print(*lst, sep = '\t')
except getopt.GetoptError :
    print('mydir.py -h -l -w')
```

Output

```
C:\>mydir  -l
data
messages
mydir
nbproject
numbers
numbersbin
numberstxt
sampledata
src
```

Problem 24.2

Define a function **show_bits( )** which displays the binary equivalent of the integer passed to it. Call it to display binary equivalent of 45.

Program

```
def show_bits(n) :
    for i in range(32, -1, -1) :
        andmask = 1 << i
        k = n & andmask
        print('0', end = '') if k == 0 else print('1', end = '')

show_bits(45)
print( )
print(bin(45))
```

Output

```
000000000000000000000000000101101
0b101101
```

Tips

- **show_bits( )** performs a bitwise and operation with individual bits of 45, and prints a 1 or 0 based on the value of the individual bit.

Problem 24.3

Windows stores date of creation of a file as a 2-byte number with the following bit distribution:

left-most 7 bits: year - 1980
middle 4 bits - month
right-most 5 bits - day

Write a program that converts 9766 into a date 6/1/1999.

Program

```
dt = 9766
y = (dt >> 9) + 1980
m = (dt & 0b111100000) >> 5
d = (dt & 0b11111)
print(str(d) + '/' + str(m) + '/' + str(y))
```

Output

```
6/1/1999
```

Tips

- Number preceded by 0b is treated as a binary number.

Problem 24.4

Windows stores time of creation of a file as a 2-byte number. Distribution of different bits which account for hours, minutes and seconds is as follows:

left-most 5 bits: hours
middle 6 bits - minute
right-most 5 bits - second / 2

Write a program to convert time represented by a number 26031 into 12:45:30.

Program

```
tm = 26031
```

```
hr = tm >> 11
min = (tm & 0b11111100000) >> 5
sec = (tm & 0b11111) * 2
print(str(hr) + ':' + str(min) + ':' + str(sec))
```

Output

```
12:45:30
```

Problem 24.5

Write assert statements for the following with suitable messages:

- Salary multiplier sm must be non-zero
- Both p and q are of same type
- Value present in num is part of the list lst
- Length of combined string is 45 characters
- Gross salary is in the range 30,000 to 45,000

Program

```
# Salary multiplier m must be non-zero
sm = 45
assert sm != 0, 'Oops, salary multiplier is 0'

# Both p and q are of type Sample
class Sample :
    pass

class NewSample :
    pass

p = Sample( )
q = NewSample( )
assert type(p) == type(q), 'Type mismatch'

# Value present in num is part of the list lst
num = 45
lst = [10, 20, 30, 40, 50]
assert num in lst, 'num is missing from lst'
```

```
# Length of combined string is less than 45 characters
s1 = 'A successful marriage requires falling in love many times...'
s2 = 'Always with the same person!'
s = s1 + s2
assert len(s) <= 45, 'String s is too long'

# Gross salary is in the range 30,000 to 45,000
gs = 30000 + 20000 * 15 / 100 + 20000 * 12 / 100
assert gs >= 30000 and gs <= 45000, 'Gross salary out of range'
```

Problem 24.6

Define a decorator that will decorate any function such that it prepends a call with a message indicating that the function is being called and follows the call with a message indicating that the function has been called. Also, report the name of the function being called, its arguments and its return value. A sample output is given below:

```
Calling sum_num ((10, 20), { })
Called sum_num ((10, 20), { }) got return value: 30
```

Program

```
def calldecorator(func) :
    def _decorated(*arg, **kwargs) :
        print(f'Calling {func.__name__} ({arg},  {kwargs})')
        ret = func(*arg, **kwargs)
        print(f'Called {func.__name__} ({arg}, {kwargs}) got ret val: {ret}')
        return ret

    return _decorated

@calldecorator
def sum_num(arg1,arg2) :
    return arg1 + arg2

@calldecorator
def prod_num(arg1,arg2) :
    return arg1 * arg2
@calldecorator
def message(msg) :
    pass
```

```
sum_num(10, 20)
prod_num(10, 20)
message('Errors should never pass silently')
```

Output

```
Calling sum_num ((10, 20), { })
Called sum_num ((10, 20), { }) got return value: 30
Calling prod_num ((10, 20), { })
Called prod_num ((10, 20), { }) got return value: 200
Calling message (('Errors should never pass silently',), { })
Called message (('Errors should never pass silently',), { }) got return
value: None
```

Exercises

[A] State whether the following statements are True or False:

(a) We can send arguments at command-line to any Python program.

(b) The zeroth element of **sys.argv** is always the name of the file being executed.

(c) In Python a function is treated as an object.

(d) A function can be passed to a function and can be returned from a function.

(e) A decorator adds some features to an existing function.

(f) Once a decorator has been created, it can be applied to only one function within the program.

(g) It is mandatory that the function being decorated should not receive any arguments.

(h) It is mandatory that the function being decorated should not return any value.

(i) Type of 'Good!' is bytes.

(j) Type of **msg** in the statement **msg = 'Good!'** is **str**.

[B] Answer the following questions:

(a) Is it necessary to mention the docstring for a function immediately below the **def** statement?

(b) Write a program using command-line arguments to search for a word in a file and replace it with the specified word. The usage of the program is shown below.

```
C:\> change  -o oldword  -n newword  -f filename
```

(c) Write a program that can be used at command prompt as a calculating utility. The usage of the program is shown below.

```
C:\> calc <switch> <n> <m>
```

Where, **n** and **m** are two integer operands. **switch** can be any arithmetic operator. The output should be the result of the operation.

(d) Rewrite the following expressions using bitwise in-place operators:

```
a = a | 3       a = a & 0x48      b = b ^ 0x22
c = c << 2      d = d >> 4
```

(e) Consider an unsigned integer in which rightmost bit is numbered as 0. Write a function **checkbits(x, p, n)** which returns True if all 'n' bits starting from position 'p' are on, False otherwise. For example, **checkbits(x, 4, 3)** will return true if bits 4, 3 and 2 are 1 in number **x**.

(f) Write a program to receive a number as input and check whether its 3rd, 6th and 7th bit is on.

(g) Write a program to receive a 8-bit number into a variable and then exchange its higher 4 bits with lower 4 bits.

(h) Write a program to receive a 8-bit number into a variable and then set its odd bits to 1.