

CSIT121

Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 8 outline

- Python built-in functions
- File I/O
- Context managers
- An alternative to method overloading
- Functions as objects

Python built-in functions

- There are numerous functions in Python that perform a task on certain types of objects.
- They usually abstract common calculations that apply to multiple types of classes.
- They accept objects with certain attributes or methods and are able to perform generic operations using those methods.
- We've used some of the build-in functions already, but let's quickly go through the important ones.

len() function

The 'len()' function

- len() function counts the number of items in some kind of container object, such as a dictionary or list.
- It is because these container objects have a method called '`__len__()`'. So when you call '`len(container_object)`', it will be executed as '`ContainerClass.__len__(container_object)`'.

```
|>>> len([1,2,3,4])  
|4
```

reversed() function

The 'reversed()' function

- The 'reversed()' function takes any sequence as input and return a copy of that sequence in reverse order
- Similar to 'len()' function, reversed calls the '___reversed___()' function on the class for the parameter.
- If the class does not have the '___reversed___()' function, reversed builds the reversed sequence itself using calls to '___len___' and '___getitem___'.
- We can override '___reversed___()' function if want to somehow customize the process.

reversed() function

```
normal_list = [1, 2, 3, 4, 5]
```

```
class CustomSequence:
    def __len__(self):
        return 5

    def __getitem__(self, index):
        return f"x{index}"
```

```
class FunkyBackwards:
    def __reversed__(self):
        return "BACKWARDS!"
```

```
for seq in normal_list, CustomSequence(), FunkyBackwards():
    print(f"\n{seq.__class__.__name__}: ", end="")
    for item in reversed(seq):
        print(item, end=", ")
```

```
>>>
```

```
= RESTART: /Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/CSIT121/Autumn_2023_Python/Python-3-Object-Oriented-Programming-Third-Edition-master/Chapter07/custom_sequence.py
```

```
list: 5, 4, 3, 2, 1,
CustomSequence: x4, x3, x2, x1, x0,
FunkyBackwards: B, A, C, K, W, A, R, D, S, !,
```

enumerate() function

- Usually, we're looping over a container in a for loop without accessing the index of each item.
- When we need to access the index, we can use the enumerate() function.
- enumerate () function creates a sequence of tuples, where the first object is the index and the second is the original item.

enumerate() function

```
import sys

filename = sys.argv[1]

with open(filename) as file:
    for index, line in enumerate(file):
        print(f"{index+1}: {line}", end="")
,
```

```
[fren@UW-XY06K990HQ examples % Python3 enumerate_lines.py enumerate_lines.py
1: import sys
2:
3: filename = sys.argv[1]
4:
5: with open(filename) as file:
6:     for index, line in enumerate(file):
7:         print(f"{index+1}: {line}", end="")
```


More useful Python built-in functions

- 'all()' and 'any()', which accept an iterable object and return True if all, or any, of the items evaluate to true (such as a non-empty string or list, a non-zero number, an object that is not None, or the literal True).
- 'eval()', 'exec()', and 'compile()', which execute string as code inside the interpreter.
- 'hasattr()', 'getattr()', 'setattr()', and 'delattr()', which allow attributes on an object to be manipulated by their string names.
- 'zip()', which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.
- And many more! See the interpreter help documentation for each of the functions listed in `dir(__builtins__)`.

```
dir(__builtins__)
['ArithmeticError', 'AssertionError',
'AttributeError', 'BaseException', 'BaseExceptionGroup', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EncodingWarning',
'EnvironmentError', 'Exception', 'ExceptionGroup', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError']
```

File I/O

- Operating systems represent files as a sequence of bytes, but not text.
- Python has wrapped the interface that operating systems provide in a sweet abstraction that allows us to work with file objects.
- We will learn some basic file-related operations, such as open, read, and write files.

Open files

A generic syntax of open function is: `open(filename, mode, encoding=None)`, where

- filename: is the name of the file you try to open/read/write. The filename is compulsory.
- The mode argument indicates the mode to use the file. Mode can be 'r' for reading only, 'w' for writing only, 'a' for appending, 'r+' for reading and writing.
- The mode argument is optional; mode 'r' (reading only) is the default mode.
- To open a binary file (video or audio file), we modify the mode string to append 'b'. So 'wb' opens a file for writing bytes, while 'rb' opens a file for reading bytes.
- The encoding argument indicates how the file is opened.
- If files are opened in text mode, then 'utf-8' is recommended unless you know what you need.

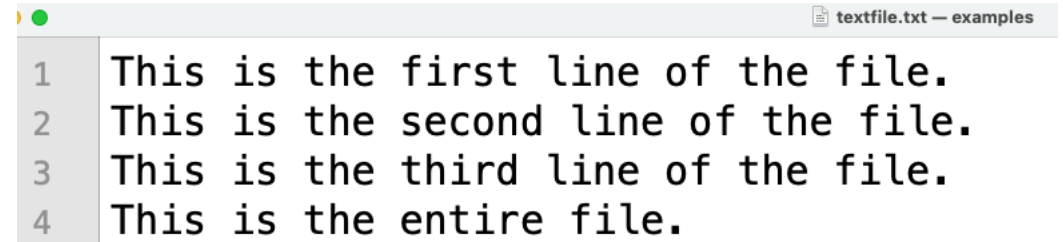
Read files

Once a file is opened, we can call the `read()`, `readline()` or `readlines()` methods to get the contents of the file.

- The `read()` method returns the entire contents of the file as string or bytes object.
- The `read()` method can also read a fixed number of bytes from file by passing an integer argument.
- The `readline()` method returns a single line from the file. We usually call the `readline()` method repeatedly to get additional lines.
- The `readlines()` method returns a list of all the lines in the file.
- It may not safe to call `read()` or `readlines()` methods to read very large files.
- The `readline()` and `readlines()` methods are also not applicable to read binary files without the newline characters, such as the image or audio files.

Read files

```
[>>> f = open('textfile.txt')
[>>> f.read(10)
'This is th'
[>>> f.readline()
'e first line of the file.\n'
[>>> f.readline()
'This is the second line of the file.\n'
[>>> f.read()
'This is the third line of the file.\nThis is the entire file.\n'
>>> f = open('textfile.txt')
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
This is the second line of the file.
This is the third line of the file.
This is the entire file.
```



```
1 This is the first line of the file.
2 This is the second line of the file.
3 This is the third line of the file.
4 This is the entire file.
```

Read files

So the best practice to read files is:

- For text files, we use a for loop to read each line at a time, and process it inside the loop body. The process will stop when we reach the end of file.
- For binary files, we'd better to read fixed-sized chunks of data in each iteration, and complete the data loading with multiple iterations.

Write files

- After we open a file, we can also call the `write()` method to write a string (or bytes for binary data) to the file.
- `write()` method returns the number of characters written.
- We can call `write()` method repeatedly to write multiple strings.
- Alternatively, we can also call the `writelines()` method to write a sequence of strings to the file.
- The `writelines()` method does not append a new line after each item in the sequence.

Write files

Write to file

```
>>> f = open('textfile.txt', 'r+')
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
This is the second line of the file.
This is the third line of the file.
This is the entire file.
>>> f.write('This is a test.\n')
16
>>> f.write('This is another test.\n')
22
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
>>> f.writelines(["See you soon!", "Over and out."])
```

Read the file after the appending:

```
>>> f2 = open('textfile.txt', 'r')
>>> print(f2.read())
This is the first line of the file.
This is the second line of the file.
This is the third line of the file.
This is the entire file.
This is a test.
This is another test.
('the answer', 42)See you soon!Over and out.
```


Close files

- After we complete file reading/writing, we should call the `close()` method to close the files.
- The `close()` method will ensure any buffered writes are written to the disk, and all resources associated with the file are released back to the operating system.
- Technically, this will happen automatically when the script exists, but it is better to be explicit and clean up by ourselves.

'with' statement

- Because exceptions may occur at any time during file I/O, we ought to wrap all method calls to a file in a try...(open/read/write files)...except...(handle exceptions)...finally...(close files regardless of whether I/O was successful) clause.
- This isn't very Pythonic, and there is a more elegant way to do it.
- If we run `dir()` on a file-like object, we will get the attributes of this object, such as

```
>>> dir(f2)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattr__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable', '_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'write_through', 'writelines']
```

'with' statement

- The `__enter__()` and `__exit__()` methods turn the file object into what is known as a context manager.
- Basically, if we use a special syntax called the 'with' statement, these methods will be called before and after nested code is executed.
- On file objects, the `__exit__()` method ensures the file is closed, even if an exception is raised.
- So we no longer need the try...finally clause, but can use the with statement.

'with' statement

```
[>>> with open('textfile.txt') as file:
[...     for line in file:
[...         print(line, end='')
[...
This is the first line of the file.
This is the second line of the file.
This is the third line of the file.
This is the entire file.
This is a test.
This is another test.
('the answer', 42)See you soon!Over and out.>>>
```

- The open call returns a file object, which has `__enter__` and `__exit__` methods.
- The returned object is assigned to the variable named `file` by the `as` clause.
- The opened file will be closed automatically when the code returns to the outer indentation level.
- The with statement can be used in the places where start up or cleanup code needs to be executed.

'with' statement

- The with statement can also be applied to any object that has the special methods, such as string objects.
- Let's create a simple context manager that allows us to construct a sequence of characters and automatically convert it to a string. We will add two special methods, i.e., `__enter__` and `__exit__` methods.
- The `__enter__` method just returns the object that will be assigned to the variable after 'as' in the 'with' statement. This is just the context manager object itself.
- The `__exit__` method accepts three arguments, i.e., type, value, and traceback. Normally, they will be given a value of None, but if any exception occurs, they can be set to other values.
- The `__exit__` method will perform any cleanup code. In our example, we create a result string by joining the characters in the string.

'with' statement

```
>>> class StringJoiner(list):
...     def __enter__(self):
...         return self
...
...     def __exit__(self, type, value, tb):
...         self.result="".join(self)
...
...
>>> import random, string
>>> with StringJoiner() as joiner:
...     for i in range(15):
...         joiner.append(random.choice(string.ascii_letters))
...
...
>>> print(joiner.result)
bDJQbwcJtFsZrMN
```

- This code constructs a string of 15 random characters.
- It appends these characters to a StringJoiner object using the 'append()' method inherited from List class.
- The __exit__ method is called automatically to append the new character to the joiner object.

Method overloading

- In OOP, method overloading refers to having multiple methods with the same name that accept different sets of arguments.
- This is very useful if we want to have a method that accepts different types of arguments.
- If we need a method to accept either an integer or a string, in non-OOP, we might need two functions, calls `add_i` and `add_s`.
- But for OOP, we can have two `add()` methods, one accepts strings and the other one accepts integers.

Default argument

- In most situations, we don't have to create a second method with a different set of arguments for method overloading but just need to make an argument optional.
- If the calling code does not supply the arguments, the default values will be used.
- Of course, the calling code can choose to override the default values by passing in different values.
- Usually, a default value of 'None', or an empty string "" or list [] will be used.

Default argument

```
def default_arguments(x,y,z,a="some string", b=False):  
    pass
```

- This example defines a function with five arguments. The first three arguments are mandatory and must be passed by the calling code. The last two parameters have default arguments supplied.

There are several ways we can call this function.

- We can provide all arguments or just the mandatory arguments in order.

```
>>> default_arguments("a string", 10, 8, "", True)  
>>>  
>>> default_arguments("a string", 5, 14)
```

- We can also use the equal sign syntax when calling a function to provide values in a different order or to skip default values.

```
>>> default_arguments("a string", 5, 14, b=True)  
>>>  
>>> default_arguments(y=5, z=14, x="a string", a="hi")
```

Keyword argument

- We can also specify an argument as a keyword argument (by placing a '*' before the argument). For example,

```
def kw_only(x, y='defaultkw', *, a, b='only'):  
    print(x, y, a, b)
```

- This function has two positional arguments 'x' and 'y', and two keyword arguments 'a', and 'b'.
- Both 'x' and 'a' are mandatory, and 'a' can only be passed as a keyword argument.
- Both y and b are optional with default value, and 'b' can only be passed as a keyword argument.

Keyword argument

```
def kw_only(x, y='defaultkw', *, a, b='only'):  
    print(x, y, a, b)
```

```
>>> kw_only('x')  
Traceback (most recent call last):  
  File "<pyshell#161>", line 1, in <module>  
    kw_only('x')  
TypeError: kw_only() missing 1 required keyword-only argum  
ent: 'a'  
>>> kw_only('x','y','a')  
Traceback (most recent call last):  
  File "<pyshell#162>", line 1, in <module>  
    kw_only('x','y','a')  
TypeError: kw_only() takes from 1 to 2 positional argument  
s but 3 were given  
>>> kw_only('x',a='a',b='b')  
x defaultkw a b  
>>> kw_only('x',a='a')  
x defaultkw a only  
>>> kw_only('x','y',a='a',b='b')  
x y a b
```

Keyword argument

- One thing to take note of with keyword arguments is that anything we provide a default argument is evaluated when the function is FIRST interpreted but not when it is called.
- This means we can't have dynamically generated default values.
- Can you guess why?

```
>>> number = 5
>>> def funky_function(number=number):
...     print(number)
...
...
>>> number = 6
>>> funky_function(8)
8
>>> funky_function()
5
>>> print(number)
6
```

Keyword argument

- This is tricky with empty containers such as lists, sets, and dictionaries.
- For example, it is common to ask calling code to supply a list that our function is going to manipulate, but the list is optional.
- We'd like to make an empty list as a default argument every time when we call the function.
- But actually, we can't do this because the function will create only one list when the code is FIRST constructed.

```
>>> def hello(b=[]):  
...     b.append('a')  
...     print(b)  
...  
...  
>>> hello()  
['a']  
>>> hello()  
['a', 'a']  
>>> hello()  
['a', 'a', 'a']
```

Variable argument lists

- Default values alone do not allow us all the flexible benefits of method overloading.
- Python allows us to write methods accepting an arbitrary number of positional or keyword arguments without explicitly naming them.
- We can also pass arbitrary container objects into such functions.
- For example, we can create a function to accept a link or list of links and download the web pages that could use such variadic arguments or varargs. The function can accept an arbitrary number of arguments.
- We do this by specifying the '*' operator in the function definition which means I'll accept any number of arguments and put them all in a list.

Variable argument lists

```
>>> def get_pages(*links):  
...     for link in links:  
...         #download the link with urllib  
...         print(link)  
...  
...  
>>> get_pages()  
>>> get_pages('http://www.google.com')  
http://www.google.com  
>>> get_pages('http://uow.edu.au', 'http://facebook.com')  
http://uow.edu.au  
http://facebook.com
```

- We can also accept arbitrary keyword arguments. These arguments arrive in the function as a dictionary.
- They are specified with two asterisks (as in `**kwargs`) in the function declaration.
- This is commonly used in configuration setups.

Variable argument lists

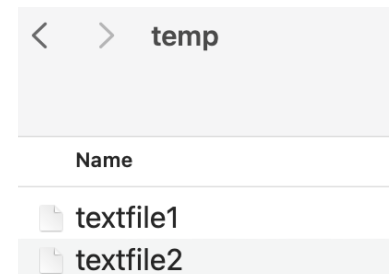
```
>>> class Options:
...     default_options = {
...         "port": 21,
...         "host": "localhost",
...         "username": None,
...         "password": None,
...         "debug": False,
...     }
...
...     def __init__(self, **kwargs):
...         self.options = dict(Options.default_options)
...         self.options.update(kwargs)
...
...     def __getitem__(self, key):
...         return self.options[key]
...
>>> options = Options(username="dusty", password="drowssap", debug=True)
>>> options['port']
21
>>> options['username']
'dusty'
>>> options['debug']
True
```


Variable argument list

- Keyword arguments are also very useful when we need to accept arbitrary arguments to pass to a second function, but we don't know what those arguments will be, i.e., in the multiple inheritance.
- This example processes an arbitrary list of files. The default behavior is to move all remaining non-keyword argument files into the target folder.
- Finally, we can supply a dictionary containing actions to perform on specific filenames.

```
def augmented_move(target_folder, *filenames, verbose=False, **specific):  
    """Move all filenames into the target_folder, allowing  
    specific treatment of certain files."""  
  
    def print_verbose(message, filename):  
        """print the message only if verbose is enabled"""  
        if verbose:  
            print(message.format(filename))  
  
    for filename in filenames:  
        target_path = os.path.join(target_folder, filename)  
        if filename in specific:  
            if specific[filename] == "ignore":  
                print_verbose("Ignoring {0}", filename)  
            elif specific[filename] == "copy":  
                print_verbose("Copying {0}", filename)  
                shutil.copyfile(filename, target_path)  
        else:  
            print_verbose("Moving {0}", filename)  
            shutil.move(filename, target_path)  
  
augmented_move("temp", "textfile1", "textfile2", "textfile3", verbose=True,  
               textfile2="copy", textfile3="ignore")
```

```
= RESTART: /Users/fren/Library/CloudStorage/OneDrive-UniversityofWollo  
ngong/MyTeaching/CSIT121/Autumn_2023_Python/examples/augmented_move.py  
Moving textfile1  
Copying textfile2  
Ignoring textfile3
```



Unpacking arguments

- Given a list or dictionary of values, we can pass those values into a function as if they were normal positional or keyword arguments.

```
>>> def show_args(arg1, arg2, arg3="Three"):
...     print(arg1, arg2, arg3)
...
...
>>> some_args = range(3)
>>> more_args = {"arg1": "One", "arg2": "Two"}
>>> show_args(*some_args)
0 1 2
>>> show_args(**more_args)
One Two Three
```

- We can use the `*` operator inside a function call to unpack a list and pass the items to the function as arguments.
- We can use the `**` operator to unpack a dictionary and pass the items to the function as arguments.

Functions are objects too

- There are numerous situations where we'd like to pass around a small object that is simply called to perform an action.
- In Python, we don't need to wrap such methods in an object because functions are already objects.
- So we can set attributes on functions (though this isn't a common activity), and we can pass them around to be called at a later date.

Functions are objects too

```
def my_function():
    print("The Function Was Called")

my_function.description = "A silly function"

def second_function():
    print("The second was called")

second_function.description = "A sillier function."

def another_function(function):
    print("The description:", end=" ")
    print(function.description)
    print("The name:", end=" ")
    print(function.__name__)
    print("The class:", end=" ")
    print(function.__class__)
    print("Now I'll call the function passed in")
    function()

another_function(my_function)
another_function(second_function)
```

```
The description: A silly function
The name: my_function
The class: <class 'function'>
Now I'll call the function passed in
The Function Was Called
The description: A sillier function.
The name: second_function
The class: <class 'function'>
Now I'll call the function passed in
The second was called
```

- We are able to set an attribute on the function.
- We are also able to see the function's name `__name__` attribute.
- We are able to access the function's class `__class__` attribute
- These all demonstrate that a function is actually an object with attributes.
- Then we call the function by using the callable syntax (the parentheses)

Using functions as attributes

- One of the interesting effects of functions being objects is that they can be set as callable attributes on other objects.
- So it is possible to add or change a function to an instantiated object.
- It is also possible to replace methods on classes instead of objects. This will change the method for all instances of that object.
- In Python, monkey patching refers to the situation that we replace or add methods at runtime in order to avoid some actions.

```
class A:
    def print(self):
        print("my class is A")

def fake_print():
    print("my class is not A")
```

```
a = A()
a.print()
my class is A
a.print = fake_print
a.print()
my class is not A
```

```
class A:
    def print(self):
        print("my class is A")
    def fake_print(self):
        print("my class is not A")
```

```
a = A()
a.print()
my class is A
a.print=a.fake_print
a.print()
my class is not A
```

Callable objects

- Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called through it were a function.
- Any object can be made callable by giving it a `__call__` method that accepts the required arguments.

```
class FoodSupply:
    def __call__(self):
        return "spam"
```

```
foo = FoodSupply()
bar = FoodSupply()
print(foo(), bar())
spam spam
```

```
class FoodSupply:
    def __init__(self, *ingredients):
        self.ingredients = ingredients
    def __call__(self):
        result = " ".join(self.ingredients) + " plus delicious spam!"
        return result
```

```
f = FoodSupply("fish", "rice")
f()
'fish rice plus delicious spam!'
```

Suggested reading

Python 3 Object-Oriented Programming

- Chapter 7: Python Object-Oriented Shortcuts

Python

- <https://www.python.org/>