Hong Kong Metropolitan University
Department of Technology

Lecture Notes

COMPS209F
Data Structures, Algorithms and Problem Solving

2022 Presentation

Table of Content

COMPS209F Data Structures, Algorithms and Problem Solving

Writing Functions 1

The concept of functions is very important in programming. Functions, which are also called subroutines, are designed to do a task for a program. Here are some examples.

Functions	***Tasks***
`input(prompt)`	Prints a prompt to the screen and reads in a string from the user. The string is returned after the function call.
`math.sin(x)`	Returns the sine of the parameter x.
`random.randrange(a, b)`	Returns a pseudo random integer ranged from a to b - 1.

Notes:

- One function should do one task.
- Call a suitable function so that a task can be done with writing just a function call.

1.1 Principles of Function Calls

In almost all programming languages, functions are provided as part of the programming tools. Programmers can use function to make their programming tasks easier.

- Programmers can use built-in functions that are provided as part of the programming language.
- Programmers can download and use functions provided by third-party. There are several sources of third party functions.
    - Commercial companies selling highly specialized libraries for programming.
    - Open source libraries from groups of volunteer programmers or consortiums of experts.
    - Libraries offered by individual programmers.
- Programmers can write their own functions.
    - These functions may be part of a program.
    - These functions may be stored for later use.
    - These functions may be shared to other programmers.

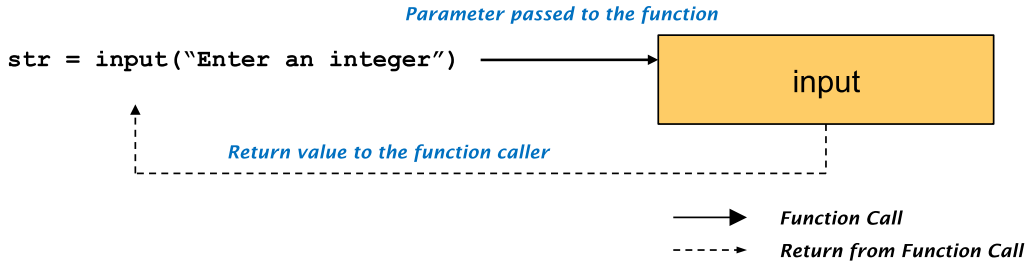
A large software development project typically has tens or hundreds of programmers working together. Each programmer will be assigned to write functions. The whole software system will be built by integrating hundreds or thousands of functions.

For better organization, related functions are bundled together. These bundles are commonly called **modules**, **libraries**, or **components**. Some large libraries, such as the renowned data science library *NumPy*, may have hundreds of functions.

1.1.1 Making a Function Call with Function Names

Function calls are made with the names of functions. A programmer should have a function in mind, and then write a function call in a program. A function call consists of the following elements:

- The **name** of the function
- The data to be passed to the function. The data are called **parameters**.
- The data returned from the function when the function call is completed. The data is called **return value**.



Notes:

- The function input() is a built-in function provided by Python.
- The parameter passed to the function is the string "Enter an integer".
- The return value to the function caller is the input actually entered by the user.

1.1.2 Binding of Function Names

Name **binding** means mapping a name to a variable or a function in a program. A program uses many names to refer to different things – modules, functions, and variables.

- A programmer may consider a name refer to one thing.
- The compiler or interpreter may consider the name refer to another thing.
- The program would likely to operate in a way deviated from the original intention.

Consider the following program. The program imports functions from three different standard modules: **random**, **statistics**, and **math**.

Example: binding_func.py

```
import random
from statistics import *
from math import sin

# built-in functions
print("This is a built-in function")  # print a message
listA = list()   # a function for creating a list


# functions from random module
x = random.randrange(1, 50)  # random.randrange refers to the randrange() of random
print(x)

# functions from statistics
listB = [1, 2, 3, 5, 6, 8]
y = mean(listB)  # this mean() comes from statistics module
print(y)
y = statistics.mean(listB)  # ERROR: but statistics is not recognized
print(y)

# function sin() from math
z = sin(3.14159)   # this sin() comes form math module
print(z)
```

Notes about calling functions imported from modules:

import random	• Import all functions from the module **random**. • The function names must be prepended with **random**. • For example, **random.randrange(a, b)**
from statistics import *	• Import all functions from the module **statistics**. • The function names must NOT be prepended with **statistics**. • For example, **mean(list)**. The function mean comes from the module **statistics**.
from math import sin	• Import only the function **sin**() from the module **math**. • The function name must NOT be prepended with **math**. • For example, **sin(3.14159)**.

A **NameError** will be raised if the function name (or variable name) is not recognizable.

- For example, **statistics.mean**() will cause **NameError**.
- The name **statistics** is not recognizable because it has not been imported explicitly.
- The following import statement has only import the functions of statistics, not the module statistics.

```
from statistics import *
```

Similarly, a **ModuleNotFoundError** will be raised if the module name is not recognizable.

The following function returns the names that have become recognizable in a program.

```
dir()

print(dir())  # print out the recognizable names
```

For the above program, the function `dir()` will return the following

```
['StatisticsError', '__annotations__', '__builtins__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'harmonic_mean', 'listA', 'listB',
'mean', 'median', 'median_grouped', 'median_high', 'median_low', 'mode', 'pstdev',
'pvariance', 'random', 'sin', 'stdev', 'variance', 'x', 'y', 'z']
```

- The following names are system names that are defined in all programs.

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__']
```

- The following names are imported functions and modules.

```
['harmonic_mean', 'mean', 'median', 'median_grouped', 'median_high', 'median_low',
'mode', 'pstdev', 'pvariance', 'random', 'sin', 'stdev', 'variance']
```

  - Module name: `'random'`
  - Function names from **`statistics`** and **`math`** modules: `'harmonic_mean', 'mean', 'median', 'median_grouped', 'median_high', 'median_low', 'mode', 'pstdev', 'pvariance', 'sin', 'stdev', 'variance'`

- The rest are variable names.

Function names can cause problems for programmers in writing a function call.

- A Python program may consist of functions from several libraries and modules.
- Clashes of function names are not uncommon.
- The libraries and modules have different programmers as authors, who have the freedom to name their modules and functions.

Alias of Module Names

An **alias** can be defined for module names. Define alias by adding **`as`** in an import statement.

```
import <module name> as <alias>
```

Consider the following example.

Example: binding_func_2.py

```
import random as ra
import numpy

# functions from random module
# now ra is an alias of random
x = ra.randrange(1, 50) # also refers to the randrange() of random

# the following would cause an error
# x = random.randrange(1, 50)

# numpy has many submodules
# numpy.char.upper() is a function changes a string into uppercase
s1 = "Python programming"
s2 = numpy.char.upper(s1)  # the module, submodules names must be included
print(s2)
```

Notes:

- The alias **`ra`** is equivalent to the random module.
- The name **`random`** is however not imported and therefore this name is not recognizable in the program.
- Use **`ra`** instead of random to refer to the module prepending the function name.

Using Functions in Sub-modules

Many large modules have used sub-modules to better organize the functions. For example, Numpy has a number of sub-modules for different types of functions.

To use functions in these sub-modules, the full path to the sub-module must be included. Note the above program again.

- There is a function **`upper(str)`** defined in the submodule **`char`** in the module **`numpy`**. The function can change a text string into uppercase.
- To call the function, the program should **`import numpy`**.
- Then the function call must include the module name and sub-module name, which is **`numpy.char.upper(s1).`**

1.1.3 Sources and Scope of Functions and Modules

The functions that a program can call must exist somewhere valid. Here are the valid locations:

- In the Python program file that is being executing.
- In a Python program file, which can be in source code form or compiled form, found in the **search path for modules**. The path normally includes the following folders (or directories):
    - The folder from which the current program file is being executing.
    - The home folder of the current user.
    - The various folders belonging the Python installation.

Assume that the current user is `andrewlui`, and the program file is `calculate.py` which is being executing in the folder `/home/andrewlui/example`. The following lists the valid locations where functions are to be found.

- The functions defined in `calculate.py`
- The functions defined in Python files sharing the same folder as `calculate.py`, which is in `/home/andrewlui/example`
- The functions defined in Python files in the home folder of the user `andrewlui`
- The functions defined in Python files in other folders defined in the search path for modules

Search Path for Modules

To know the search path for modules, look at the content of `sys.path`, which is a list. Add the following to your program to look at the path.

```
print(sys.path)
```

Something similar to the following should be printed.

```
['/home/andrewlui/example, '/home/andrewlui', '/home/andrewlui/anaconda3/bin',
'/home/andrewlui/anaconda3/lib/python37.zip',
'/home/andrewlui/anaconda3/lib/python3.7',
'/home/andrewlui/anaconda3/lib/python3.7/lib-dynload',
'/home/andrewlui/anaconda3/lib/python3.7/site-packages']
```

Additional paths may be added to the system's search path for module. Simply use the append method to add a new folder that contains Python files for import.

```
>>> sys.path.append('/home/andrewlui/mypython/')
```

Installation of Third-Party Modules

A Python programmer wishes to use a third-party module should first install it. This installation means adding the relevant Python files to one of the locations where the search path has included.

Normally, third-party module providers should include information of how to install the module. These are some of the common methods.

- Download the module as a ZIP file or Python files. Add the files to the same folder as your Python programs.
- Download the module as a ZIP file or Python files. Add the files to a dedicated folder. Modify the `sys.path` to include the dedicated folder.
- Use the package manager for Python called **`pip`**. The program should be executed from a command prompt of the operating system. For example to install the renowned module for deep machine learning called tensorflow.

  ```
  > pip install tensorflow
  ```

1.2 Writing Your Functions

Like for structures and while structures, function definition should follow a particular structure.

1.2.1 Basic Structure of Function Definition

Writing a function should follow the structure below.

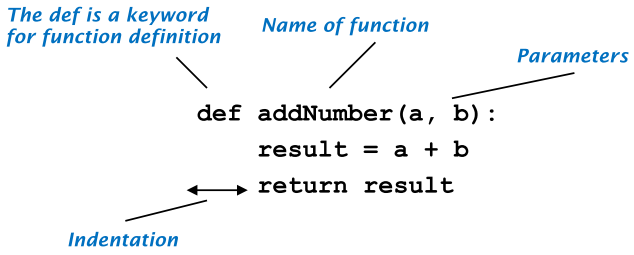
Example: simple_func_1.py

```
def addNumber(a, b):
    result = a + b
    return result
```

Notes:

- The **`def`** is a keyword meaning that this block is a function definition.
- The **`addNumber`** following `def` is the name of the function.

- A pair of round brackets should follow the function name.
    - Within the round brackets there are optional parameter variables.
    - Parameter variables are data receiving from the function call.
    - The above function has two parameters **`a`** and **`b`**.
- The Python code that belongs to the function is indented.



Purpose of the Function

A function should be characterised with the following:

- The one task that it is designed to do.
- The data sent to the function for doing the task. This is called **data-in**.
- The data returned to the caller after completing the task. This is called **data-out**.

The example function should be characterised as below.

Name of the function	**`addNumber`**
Purpose or task of the function	This function adds two numbers and returns the sum.
Data-in	The two numbers to be added are found in parameters **`a`** and **`b`**.
Data-out	The sum of the two numbers

Calling the Function From Same File

The following shows the whole content of the Python file.

Example: simple_func_1.py

```
def addNumber(a, b):
    result = a + b
    return result

x = 10
y = 20
z = addNumber(x, y)
print("10 + 20 =", z)
```

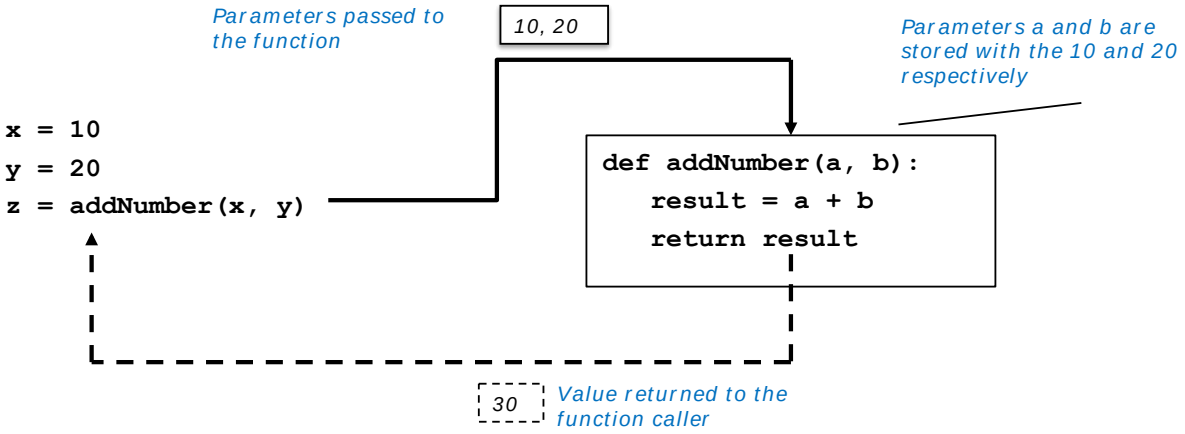
Notes:

- The Python file contains a function definition.
- It also contains code to be executed if the file or module is executed.
- The code contains a function call to `addNumber`.

```
z = addNumber(x, y)
```

    - The function call passes two values to `addNumber`. The two values are 10 and 20. These two values come from **`x`** and **`y`**.
    - When the function call is completed, there is a return value 30 that is assigned to variable **`z`**.

- The function definition must appear before the function call.
    - The declaration of function must be executed first.
    - Python must know the function (i.e. the function name) when the function call is made.



- Executing this file or module will print the following output to the screen

```
10 + 20 = 30
```

Calling the Function From Another Program File

The function **`addNumber`** can be called from another program file.

Create another program file called `test_func_1.py` and import the module file `simple_func_1`. The two program files are stored in the same folder.

Example: test_func_1.py

```
import simple_func_1

b = 300
result = simple_func_1.addNumber(200, b)
print("200 + 300 =", result)
```

```
10 + 20 = 30
200 + 300 = 500
```

Notes:

- The program uses an import statement to include the content of `simple_func_1.py`.
    - The import operation executes the content of `simple_func_1.py`.
    - The operation includes the definition of the function **`addNumber`**.
    - It also executes the content below.

```
x = 10
y = 20
z = addNumber(x, y)
print("10 + 20 =", z)
```

- The first line of the output therefore shows below – an evidence that `simple_func_1.py` has been executed.

```
10 + 20 = 30
```

- The rest of the test_func_1.py was then executed – which prints the following output to the screen.

```
200 + 300 = 500
```

Conditional Code Execution During Module Import

When a module is imported during the execution of a program file, the content of the module is executed.

- In most cases, the target of import limits to function definition.
- To prevent the execution of content that is not function definition, a programmer can add a selection structure.

Example: simple_func_2.py

```
def addNumber(a, b):
    result = a + b
    return result

# start executing here if this Python file or module is executed
# the following is not executed if this module is imported.
print(__name__)
if __name__ == "__main__":
    x = 10
    y = 20
    z = addNumber(x, y)
    print("10 + 20 =", z)
```

Notes:

- The system variable __name__ contains the string `"__main__"` if this file is directly executed.
- It contains `"simple_func_2"` if this Python file is executed during import.
    - In such case the body of the `if` structure is not executed.

The following output is shown if `simple_func_2.py` is directly executed.

```
__main__
10 + 20 = 30
```

Consider the following program, which imports `simple_func_2`.

Example: test_func_2.py

```
import simple_func_2

b = 300
result = simple_func_2.addNumber(200, b)
print("200 + 300 =", result)
```

```
simple_func_2
200 + 300 = 500
```

Notes:

- The output shows that the if condition in `simple_func_2` is `False`.
- That if body part of the code is skipped.

> ❖ ***Parameters and Arguments***
>
> This lecture note uses the term parameters to describe both the values passed in function calls, and the parameter variables inside functions.
>
> Some textbooks use the term arguments to describe the values passed in function calls.
>
> Readers are reminded of the different use of the terms.

1.2.2 Example: Factorial

The factorial of an integer N is defined as:

$$factorial(N) = 1 \times 2 \times \cdots \times N$$

This is a common mathematical operation. It would be useful to define a function for repeated use.

Name of the function	**`factorial`**
Purpose or task of the function	This function calculates the factorial of an integer N >= 0
Data-in	The integer N
Data-out	The factorial of N (an integer)

The following shows an implementation.

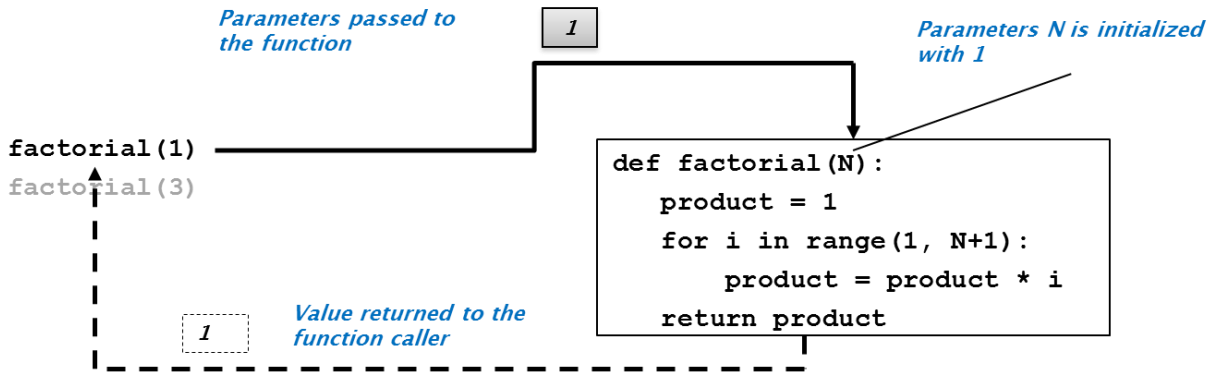
Example: factorial.py

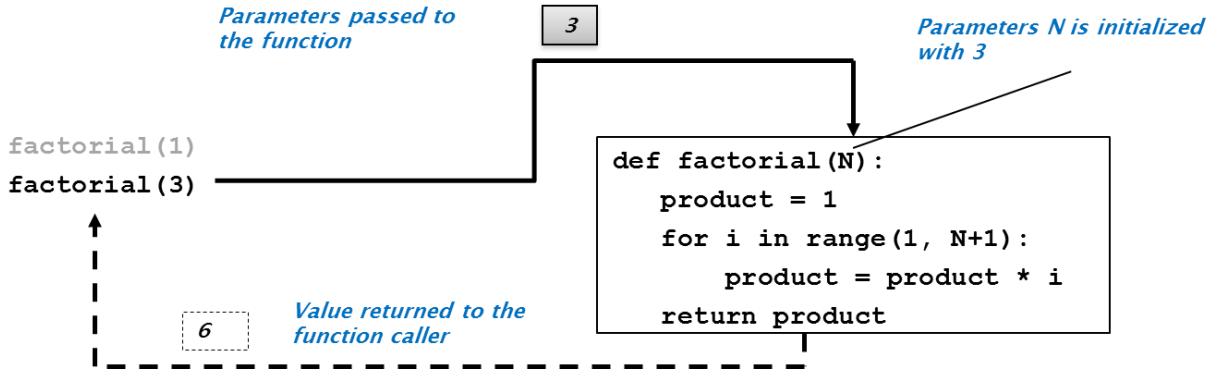
```
def factorial(N):
    product = 1
    for i in range(1, N+1):
        product = product * i
    return product

# for testing
if __name__ == "__main__":
    print("Factorial(1) =", factorial(1))
    print("Factorial(3) =", factorial(3))
    print("Factorial(10) =", factorial(10))
```

```
Factorial(1) = 1
Factorial(3) = 6
Factorial(10) = 3628800
```

Notes:

- The function has a single parameter with the name N. This is called a **parameter local variable**.
  - A parameter local variable is created when the function is called.
  - It is initialized with the value passed with the function call.
  - Normally using a variable without assigning a value first causes an error. However, Python has already initialized parameter local variables at function calls. No error occurred in the above program.
- The function has created a variable named **`product`**. This is called a **general local variable**.
- **Local variables** are variables created inside a function.
  - Local variables are created at the beginning of function calls.
  - Local variables are destroyed at the completion of function calls.





- The local variable product, at the end of the for loop, contains the result of factorial calculation.
- The `return product` is called a **return statement**. This statement specifies which value is to be sent back to the caller.

1.2.3 Example: Factorial with Error Handling

Factorial of a negative N is undefined.

Consider the factorial function implemented in the previous section. If the function is called with a negative parameter, the function returns 1. This is incorrect.

```
>>> factorial(-1)
1
```

A function should check for and handle error parameters. There are two common methods.

- Return a special value indicating an error.
- Raise an exception.

Error Handling Method 1: Return a Special Value

The following function implementation returns the **`None`** value for error parameters.

Example: factorial.py

```
def factorial2(N):
    if N < 0:
        return None
    product = 1
    for i in range(1, N+1):
        product = product * i
    return product

# for testing
if __name__ == "__main__":
    print("Factorial2(-1) =", factorial2(-1))
```

```
...
Factorial2(-1) = None
```

Notes:

- The error parameter checking code should be placed right at the beginning of function definition.
- The error condition is `N < 0`.

Error Handling Method 2: Raise an Exception

The following function implementation raises ValueError for error parameters.

Example: factorial.py

```
def factorial3(N):
    if N < 0:
        raise ValueError('Negative N not allowed')
    product = 1
    for i in range(1, N+1):
        product = product * i
        return product

# for testing
if __name__ == "__main__":
    print("Factorial2(-1) =", factorial3(-1))
```

```
...
Traceback (most recent call last):
  File "C:\...\factorial.py", line 43, in <module>
    print("Factorial3(-1) =", factorial3(-1))
  File "C:\...\factorial.py", line 17, in factorial3
    raise ValueError('Negative N not allowed')
ValueError: Negative N not allowed
```

Notes:

- A message “Negative N is not allowed” is added to the **`ValueError`** to describe the error.
- The exception will cause the program to terminate, unless the caller has a **`try-except`** block to handle the exception.

1.2.4 Example: List Containing the Same Element

This section describes a function that checks if a list contains the same element.

The following shows a few examples.

Example: A List with Same Elements

```
[1, 1, 1, 1, 1, 1, 1]
```

Example: A List does not have the Same Elements

```
[1, 1, 1, 1, 2, 1, 1]
```

Example: A List with Same Elements (List of Lists)

```
[[2, 2, 2], [2, 2, 2], [2, 2, 2]]
```

The function should return **`True`** if all elements in a list are the same.

Name of the function	**`sameElements`**
Purpose or task of the function	This function checks if all elements in a list are the same
Data-in	One parameter which is the list to be checked
Data-out	**`True`** or **`False`**

Error Handling: The Parameter Must be a List

Error checking is needed to make sure that the parameter passed from the caller is a list. Use the type() built-in function to get the list of a variable or object.

Example

```
def sameElements(alist):
    if type(alist) != list:
        raise ValueError('Parameter should be a list')
    ...
```

Implementation

The following shows an implementation. The function uses a for loop to traverse all elements in a list. For each element, match it with the first element of the list (`alist[0]`).

Example: list_func.py

```
# returns True if all elements in alist are the same
def sameElements(alist):
    if type(alist) != list:
        raise ValueError('Parameter should be a list')
    if len(alist) == 0:
        return False    # an empty list is assumed to be False
    for value in alist:
        if value != alist[0]:
            return False
    else:
        return True
```

Notes:

- The function should handle the special case for empty lists.
    - For empty list, the function should return `False`.
- The function returns `False` immediately if any element is not the same as `alist[0]`.

❖ *Exit Points of Functions*

The default exit point is the end of function. When the execution reaches the end of the function body, the function call is completed.

The optional return statement can define additional exit points. Similarly, raising an exception is also an exit point.

```
def sameElements(alist):

    if type(alist) != list:
        raise ValueError('Parameter should be a list')
    if len(alist) == 0:
        return False
    for value in alist:
        if value != alist[0]:
            return False
    else:
        return True
```

Exit Point #1

Exit Point #2

Exit Point #3

Exit Point #4

The Default Exit Point

If the function is designed to return a value, then a return statement with a value is necessary.

Black-Box Testing

The function should be tested to ensure it is working correctly. **Black-box testing** means calling the function with some parameters, and then check if the outcomes are as expected.

Black-box testing should be carried out systematically.

Description of Test Cases	***Parameters***	***Expected Outcome***
Non-List Parameters	`None`	ValueError raised
Non-List Parameters	`5` *(an integer)*	ValueError raised
Empty List	`[]`	Returns `False`
List with One Element	`[1]`	Returns `True`
List with Same Elements	`[1, 1, 1]`	Returns `True`
List of Lists with Same Elements	`[2, 2, 2], [2, 2, 2], [2, 2, 2]`	Returns `True`
List with Different Elements	`[1, 2, 1]`	Returns `False`

The above test cases are implemented in a program with lots of function calls to `sameElements`.

Example: list_func.py

```
# for testing
if __name__ == "__main__":
    # test error paramters
    try:
        print("sameElements(None) =", sameElements(None))
        # print("sameElements(5) =", sameElements(5))
    except ValueError as e:
        print("ERROR:", e)

    # test different lists
    print("sameElements([]) =", sameElements([]))
    print("sameElements([1]) =", sameElements([1]))
    print("sameElements([1, 1, 1]) =", sameElements([1, 1, 1]))
    print("sameElements([1, 2, 1]) =", sameElements([1, 2, 1]))
    # test list of lists with same elements
    listA = [[2, 2, 2], [2, 2, 2], [2, 2, 2]]
    print("sameElements({}) = {}".format(listA, sameElements(listA)))
```

```
ERROR: Parameter should be a list
sameElements([]) = False
sameElements([1]) = True
sameElements([1, 1, 1]) = True
sameElements([1, 2, 1]) = False
sameElements([[2, 2, 2], [2, 2, 2], [2, 2, 2]]) = True
```

Notes:

- The outcomes of calling the function with different parameters match the expectation.
- The function can be regarded as correct (i.e. until a programming error is found).

1.2.5 Example: List Containing the Same Given Element

This section describes modifying the function so that it checks if the list contains only the given target parameter.

- Two parameters, a list and a target value, are given to the function.
- The function should return `True` if all elements in the list are same as the given element.

Name of the function	**`sameGivenElements`**
Purpose or task of the function	This function checks if all elements in a list are same as the given target parameter
Data-in	Two parameters: (1) a list (2) the target value
Data-out	**`True`** or **`False`**

Implementation

The following shows an implementation.

Example: list_func.py

```
# returns True if all elements in alist are x
def sameGivenElements(alist, x):
    if type(alist) != list:
        raise ValueError('Parameter should be a list')
    if len(alist) == 0:
        return False    # an empty list is assumed to be False
    for value in alist:
        if value != x:
            return False
    else:
        return True
```

Notes:

- The bold lines are the changed lines.
- The elements in the list are compared to the parameter **`x`** instead of **`alist[0]`**.

Black-Box Testing

The function was tested with the following test cases.

Example: list_func.py

```
# for testing
if __name__ == "__main__":
    # test different lists with different given elements
    print("sameGivenElements([], 1) =", sameGivenElements([], 1))
    print("sameGivenElements([1], 1) =", sameGivenElements([1], 1))
    print("sameGivenElements([1], 2) =", sameGivenElements([1], 2))
    print("sameGivenElements([1, 1, 1], 1) =", sameGivenElements([1, 1, 1], 1))
    print("sameGivenElements([1, 1, 1], 2) =", sameGivenElements([1, 1, 1], 2))
    print("sameGivenElements([1, 2, 1], 1) =", sameGivenElements([1, 2, 1], 1))
    # test list of lists
    listA = [[2, 2, 2], [2, 2, 2], [2, 2, 2]]
    print("sameGivenElements({}, 2) = {}".format(listA, sameGivenElements(listA, 2)))
    print("sameGivenElements({}, [2, 2, 2]) = {}".format(listA, sameGivenElements(listA,
[2, 2, 2])
```

```
sameGivenElements([], 1) = False
sameGivenElements([1], 1) = True
sameGivenElements([1], 2) = False
sameGivenElements([1, 1, 1], 1) = True
sameGivenElements([1, 1, 1], 2) = False
sameGivenElements([1, 2, 1], 1) = False
sameGivenElements([[2, 2, 2], [2, 2, 2], [2, 2, 2]], 2) = False
sameGivenElements([[2, 2, 2], [2, 2, 2], [2, 2, 2]], [2, 2, 2]) = True
```

1.2.6 Example: One Function, Two Modes of Checking

The previous two functions, `sameElements` and `sameGivenElements`, are very similar in their purposes.

- The two functions represent two modes of checking.
- `sameElements` checks if all elements in a list are the same.
- `sameGivenElements` checks if all elements in a list are as same as a target value.
    - It has one more parameter than `sameElements`.
    - The additional parameter is the target value.

Python provides a way to specify a **default value** to a parameter. A parameter with a default value is optional – the caller can leave it out in the parameter list.

The following new definition of a function has combined `sameElements` and `sameGivenElements` into one.

Name of the function	**`sameElements`**
Purpose or task of the function	This function checks if all elements in a list are the same. Optionally the function can check if all the elements match a target value.
Data-in	Parameter #1: the list to be checked Parameter #2: the target value (optional)
Data-out	**`True`** or **`False`**

Parameters with Default Values

Specifying a default value for a parameter is easy – just add an assignment operation with a value after the parameter variable name.

Example

```
def sameElements(alist, x=None):
    ...
```

Notes:

- The parameter **`x`** has a default value of **`None`**.
- If only one parameter is provided in the function call to **`sameElements`**, then the parameter variable **`x`** is initialized to **`None`**.

Example

```
sameElements([1, 1, 1])   # one parameter, x is defaulted to None

sameElements([1, 1, 1], 2)   # the caller provides the second parameter, x is 2
```

Implementation

The following shows an implementation based on the second parameter with a default value.

Example: list_func_2.py

```
# returns True if all elements in alist are the same AND
# the elements equal to x if it is given
def sameElements(alist, x=None):
    if type(alist) != list:
        raise ValueError('Parameter should be a list')
    if len(alist) == 0:
        return False
    if x is None:    # if x is not provided, then set x to be alist[0]
        x = alist[0]
    for value in alist:
        if value != x:
            return False
    else:
        return True
```

Notes:

- The for loop is the same as the version in **`sameGivenElements`**.
    - The elements in the list are checked against the variable **`x`**.
- In this function, the variable **`x`** is either provided through parameter of function calls.
- If function call has no second parameter, then **`x`** is assigned **`alist[0]`**.
    - Making the comparison same as the version in the original **`sameElements`**.

Black-Box Testing

The function was tested with the following test cases.

Example: list_func.py

```
# for testing
if __name__ == "__main__":
    print("sameElements([]) =", sameElements([]))
    print("sameElements([], 1) =", sameElements([], 1))
    print("sameElements([1]) =", sameElements([1]))
    print("sameElements([1], 1) =", sameElements([1], 1))
    print("sameElements([1], 2) =", sameElements([1], 2))
    print("sameElements([1, 1, 1]) =", sameElements([1, 1, 1]))
    print("sameElements([1, 2, 1]) =", sameElements([1, 2, 1]))
    print("sameElements([1, 1, 1], 1) =", sameElements([1, 1, 1], 1))
    print("sameElements([1, 1, 1], 2) =", sameElements([1, 1, 1], 2))
    print("sameElements([1, 2, 1], 1) =", sameElements([1, 2, 1], 1))
    # test list of lists with same elements
    listA = [[2, 2, 2], [2, 2, 2], [2, 2, 2]]
    print("sameElements({}) = {}".format(listA, sameElements(listA)))
    print("sameElements({}, 2) = {}".format(listA, sameElements(listA, 2)))
        print("sameElements({}, [2, 2, 2]) = {}".format(listA, sameElements(listA, [2,
2, 2])))
```

```
sameElements([]) = False
sameElements([], 1) = False
sameElements([1]) = True
sameElements([1], 1) = True
sameElements([1], 2) = False
sameElements([1, 1, 1]) = True
sameElements([1, 2, 1]) = False
sameElements([1, 1, 1], 1) = True
sameElements([1, 1, 1], 2) = False
sameElements([1, 2, 1], 1) = False
sameElements([[2, 2, 2], [2, 2, 2], [2, 2, 2]]) = True
sameElements([[2, 2, 2], [2, 2, 2], [2, 2, 2]], 2) = False
sameElements([[2, 2, 2], [2, 2, 2], [2, 2, 2]], [2, 2, 2]) = True
```

Notes:

- This new version of `sameElements` can accept one parameter or two parameters.
- They represent two modes of checking.

1.3 Variables of Functions and Scope Rules

Functions are defined in programs. However, the code inside the function may not be used at all. If the functions are not called, then the relevant code is never executed.

Function calls involve some important concepts of programming. This section discusses the use of variables inside and outside functions.

- Can variables be created inside functions?
- Can the variables accessed from outside (i.e. out of the function)?
- Can variables share the same name?

1.3.1 Local Variables and Global Variables

Global variables are variables created outside of a function (i.e. the main program).

Local variables are variables created within a function.

There are two types of local variables:

- General local variables. They are created and initialized explicitly.
- Parameter local variables. They are initialized with the function call's parameters.

Example: variable_1.py

```
apple = 1  # creates a global variable apple
print(apple)
if apple == 1:
    orange = False  # creates a global variable orange
    print(orange)
print(orange)

def functionA(tea, coffee):
    print(tea, coffee)
    mokka = 2
    print(mokka + 10)

drink = 5
functionA(drink, 2)
print(apple, orange, drink)
```

Life Cycle of Local Variables

Local variables have a **life cycle**:

- Parameter local variables are created at the beginning of function call.
    - Their values are initialized with the parameters passed with the function call.
- Because such initialization is inside

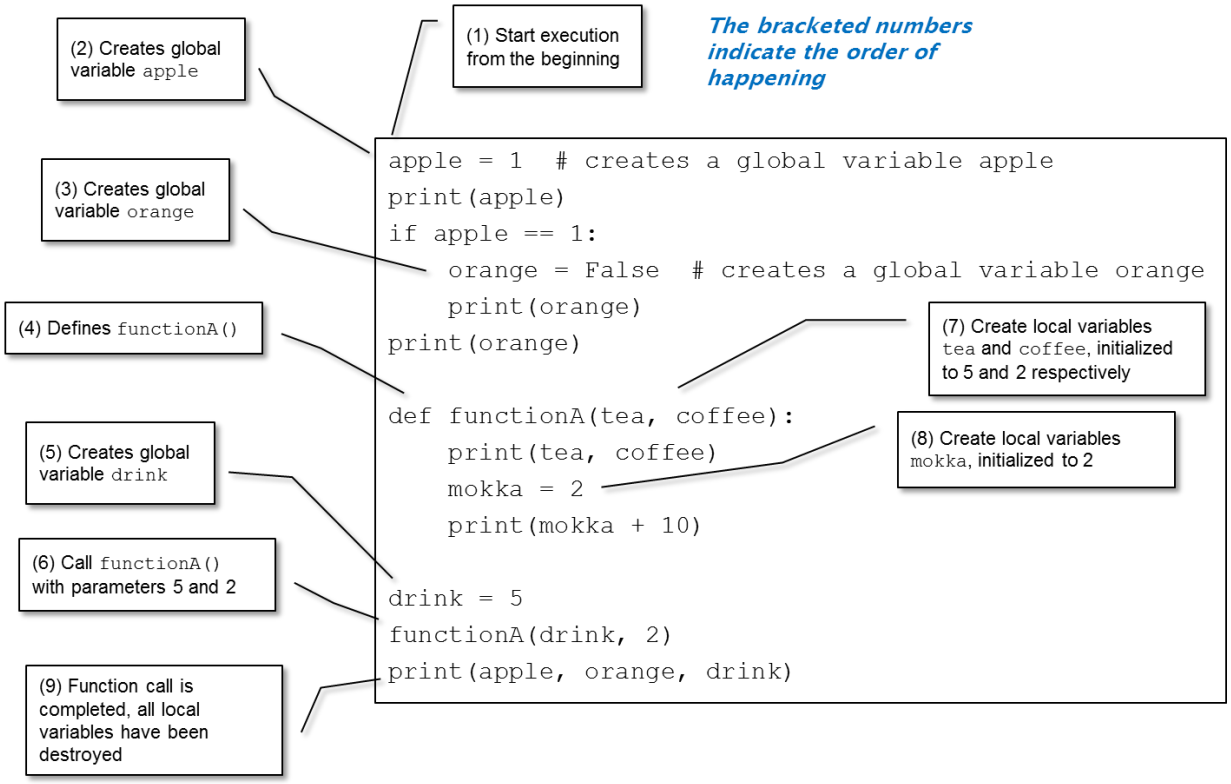
Global variables remain as long as program execution has not finished.

1.3.2 Execution Model With Respect to Variable Creation

Executing a Python file or module normally starts from the first line of the file. The execution then continues from one line to the next line, subject to control structures such as selection and repetition structure.

The following diagram shows when the variables (and function) of the above program are created.

- The local variables inside functions may only be created when the function is called.
- If a function is only defined but never get called, any related local variables are never created.



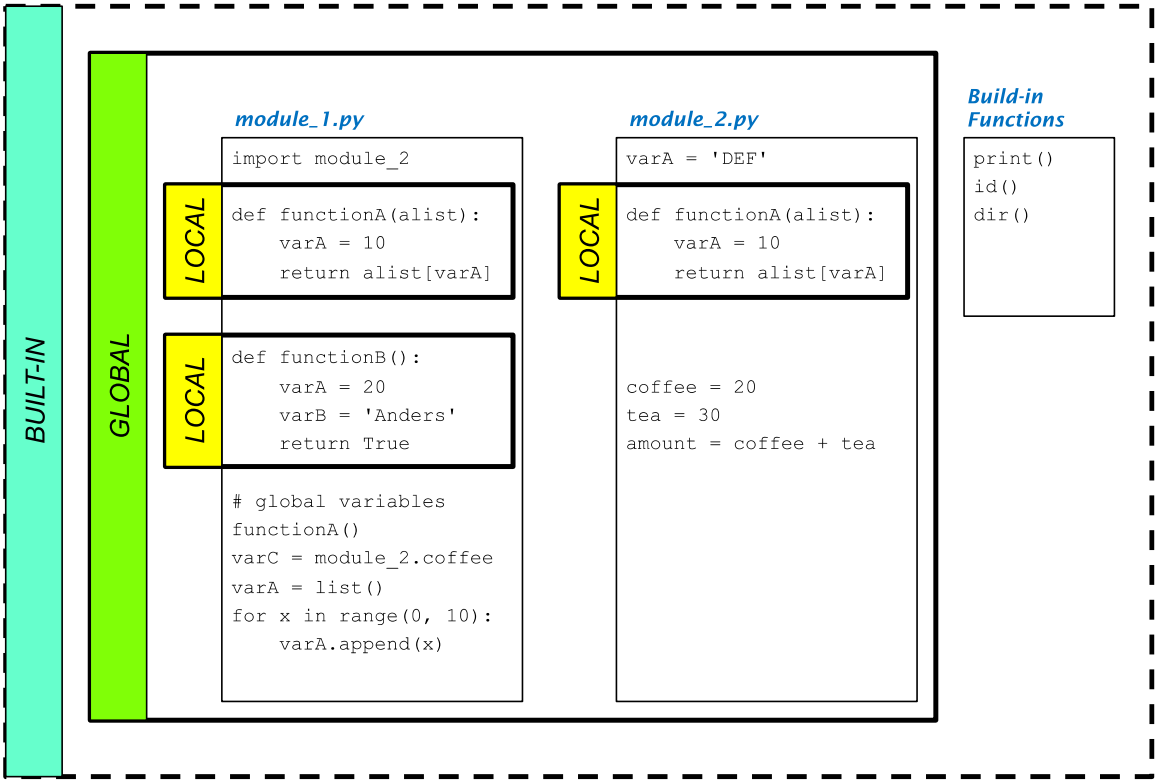
1.3.3 Variable Scoping Rules and Namespaces

The locations of the creation and the use of a variable are of great significance in programming.

- Naming: Determines the validity of variable names.
- Accessibility: Determines whether a variable is allowed to use.
- Binding: Determines which variable is used in the case of name clashes.

There are three types of locations, or **scopes**, in Python programs.

- Local scope: within a function.
    - Each function has its own local scope.
    - One function's local scope is different from another function's local scope.
- Global scope: outside a function and include any imported modules and files.
- Built-in scope: everywhere in the program.
    - This scope is Python defined.



Naming Rules

Naming rules govern whether variable names can be the same.

- Local variables of the same local scope cannot have the same name.
- Global variables of the same module cannot have the same name.
    - Global variables of another module can be used after the import of the module. Their names must be prepended with the module name.

Example 1: NO ERROR

```
1
2
3
```

```
coffee = 10


coffee = 20
```

The variable **`coffee`** on line 1 was assigned for the first time and was created in the global scope. The variable **`coffee`** on line 3 was the same variable created on line 1.

Example 2: NO ERROR

```
1
2
3
4
```

```
def functionA():
    coffee = 10


    coffee = 20
```

When this function is called, the variable **`coffee`** on line 2 was assigned for the first time and was created in the local scope. The variable **`coffee`** on line 4 was the same variable created on line 2.

Example 3: NO ERROR

```
1
2
3
4
5
```

```
def functionA():
    coffee = 10


coffee = 20
functionA()
```

The variable **`coffee`** on line 4 was assigned for the first time and was created in the global scope. Then **`functionA()`** is called. The variable **`coffee`** on line 2 was assigned for the first time in the local scope and was created. Then the function call is completed, the local scope's variable **`coffee`** will be destroyed.

Example 4: NO ERROR

module_1.py

```
1  import module_2
2
3  coffee = 20
4  print(coffee) # 20
5  print(module_2.coffee) # Betsy
```

module_2.py

```
1
2  coffee = 'Betsy'
3
4
5
```

The file `module_1.py` was executed. The file `module_2.py` was imported and its code executed. The variable **`coffee`** on line 5 of `module_2` was created in the global scope (i.e. under `module_2`).

The execution of `module_1.py` continued. The variable **`coffee`** on line 3 of `module_1` was created in the global scope. The use of variable **`coffee`** in line 4 refers to the variable coffee of `module_1`. Then `module_2.coffee` in line 5 refers to the global scope coffee in `module_2`.

Accessibility Rules

Accessibility rules govern whether a variable defined in one scope can be used in another scope.

- A local scope variable can only be accessed within the same local scope.
    - Other local scopes and the global scope cannot access that variable.
- A global scope variable can be accessed in global scope and in all local scopes.

Example 1: ERROR

```
1  def functionA():
2      coffee = 10
3      print(coffee + 2)
4
5  functionA()
6  print(coffee)
```

The **`functionA()`** was defined in line 1 and called in line 5.

The use of **`coffee`** in line 6 is an error.

Binding Rules

Sometimes a name may refer to more than one variable (and functions).

- In a local scope, the variables created in the same local scope and the global scopes are accessible.
- In addition, naming rules allow two variables, one in a local scope and another in the global scope, share the same name.
- A rule is needed to resolve the ambiguity.

Binding rules govern which variable is referred to when a variable name is used. There is only one rule.

- Local scope always takes precedent over global scope.

Example 1: NO ERROR

```
1  def functionA():
2      coffee = 10
3      print(coffee) # 10
4
5  coffee = 20
6  functionA()
```

The name **`coffee`** in line 3 may refer to (1) the global variable **`coffee`** created in line 5 or (2) the local variable **`coffee`** created in line 2.

The local variable takes precedence, and 10 is printed to the screen.

Redefinition of Names

A name binding may be over-written by redefinition.

- The name of a built-in function is used for creation of variables.

Example 1

```
1
2
3  # redefinition of print
4  print = 10
5
6  print("ABC")
```

A variable called print is created in line 4. This has overwriten the name `print`. Line 6 causes an error because the name print is no longer a function.

- A function definition uses a name that has been defined before. The latest definition is the one that is used in function call.

Example 2

```
1  def input(s):
2      print("I do not do input")
3      return None
4
5  def input(s):
6      print("This is latest")
7      return 1
8
9  num = input("Enter a number:")
10 print(num)
```

Line 1 to 3 defines a function named `input`, which has overwritten the built-in function `input`.

Line 5 to 7 defines another function also named `input`, which has overwritten the one in Line 1 to 3.

Line 9 contains a function call that uses the latest definition of `input`.

Example

The following shows an example of naming, accessibility, and binding rules.

module_1.py

```
1  import module_2
2
3  def functionA(alist):
4      varA = 10
5      return varA + len(alist)
6
7  def functionB():
8      varA = 20
9      varB = 'Anders'
10     return True
11
12 # global variables
13 varC = module_2.coffee
14 varA = 0
15 listA = list()
16 functionA(listA)
17
18 module_2.functionA(listA)
19 print(varC, varA)
20 for x in range(0, 10):
21     listA.append(x)
```

module_2.py

```
1  varA = 'DEF'
2
3  def functionA(alist):
4      varA = 10
5      return len(alist)
6
7  coffee = 20
8  tea = 30
9  amount = coffee + tea
```

Notes:

- For `module_1.py`:
  - The use of `varA` on line 5 refers to the local variable `varA` created on line 4.
  - The use of `alist` on line 5 refers to the local parameter variable `alist` created on line 3.
  - The use of `module_2.coffee` on line 13 refers to the global variable `coffee` created on line 7 in `module_2`.
  - The use of `varA` on line 19 refers to the global variable `varA` created on line 14.

- For `module_2.py`:
    - The use of `alist` on line 5 refers to the parameter local variable `alist` created on line 3.
    - The use of `coffee` on line 9 refers to the global variable `coffee` created on line 7.

1.4 Parameter Passing and Return Values

For most functions to do the designed task, they need information from the caller. When they finish the task, they should send back information to the caller.

- How parameters are sent to functions?
- How return values are sent back to callers?

1.4.1 Specification of Parameters

Parameters are part of function definition.

- The number of parameters.
    - A function may have zero parameter.
- The names of the parameters.
- The parameters in Python have no type definition.
    - Many other languages require parameters to have a specified type.

Python also provides some advanced specifications for parameters.

- Optional default values for parameters.
    - These parameters may be omitted in function calls.
- Variable number of parameters
    - The function can accept any number of parameters.

Examples of Parameter Specifications

Example #1	*Example #2*	*Example #3*
`def addNumbers(a, b, c, d):`	**`def mean(list):`**	**`def showall():`**
This function accepts four parameters, with names **`a, b, c`** and **`d`**.	This function accepts one parameter called **`list`**.	This function does not accept any parameter.

Example #4	
`def check(list, x=None):`	
This function accepts at least one parameter, with the name **`list`**. There is an optional second parameter called **`x`**.	

1.4.2 Passing Parameters with Function Calls

A function call should pass the number of parameters, as required by the function definition.

- For example, if the function accepts four parameters, then the function call should include four parameters.

Example

```
def addNumbers(a, b, c, d):
    print(a, b, c, d)
    ...
...
x = 2
y = 5
addNumbers(1, x, 10, y)
```

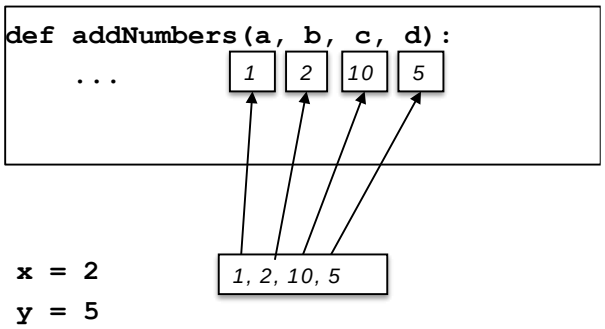
- Both variables and constants can be a parameter passed in a function call. If the parameter is a variable, actually its value is passed.
    - In the above example, the four parameters passed are 1, 2, 10, 5 respectively for **`a, b, c`** and **`d`**. These four integers are printed in the function's print statement.

Positional Parameter Passing and Keyword Parameter Passing

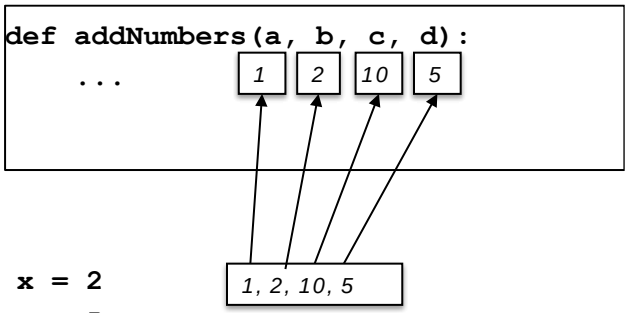
Python offers two main ways to pass parameters in function calls.

These two ways differ in how the parameters (i.e. arguments) passed are mapped (i.e. copied) to the parameter variables in functions.

- **Positional parameters**. The parameters are copied to the parameter variables according to the order and position of the parameters.
    - The first parameter is copied to the first parameter variable. The second parameter is copied to the second parameter variable, and so on.



- **Keyword (named) parameters**. The parameters are copied to the parameter variables according to the names given in the function call.
    - The names refer to the parameter variable names in the function definition.



The following shows some valid examples of using positional parameters, keyword parameters, and a mix of the two.

Example: parameter_pass_1.py

```
def addNumbers(a, b, c, d):
    print(a, b, c, d)
    result = a + b + c + d
    return result

x = 2
y = 5
# positional parameter passing
addNumbers(1, x, 10, y)

# keyword parameter passing
addNumbers(a=1, d=y, c=10, b=x)

# keyword parameter passing
addNumbers(d=y, c=10, a=1, b=x)

# mixed ways of parameter passing
addNumbers(1, x, c=10, d=y)
addNumbers(1, x, d=y, c=10)
```

```
1 2 10 5
1 2 10 5
1 2 10 5
1 2 10 5
1 2 10 5
```

Notes:

- All five function calls passed the same set of data to the parameter variables in the function.
- Positional parameters follow exactly the same order as the parameter list in the function definition.
- Keyword parameters can be in any order.
- In mixed parameters, position parameters must come before keyword parameters.

Optional Parameters

Optional parameters have a default value defined.

- If the parameter is omitted, then the default value is used.
    - In the following example, the first function call provides all four parameters. The parameter local variable **`d`** is assigned 5.
    - The second function call provides only three parameters. The parameter local variable **`d`** is set to default value 0.

Example

```
def addNumbers(a, b, c, d=0):
    print(a, b, c, d)
    ...
...
x = 2
y = 5
addNumbers(1, x, 10, y)
addNumbers(1, x, 10)
```

- The optional parameter may be provided with either positional or keyword way.

The following shows some valid examples of using positional parameters, keyword parameters, and a mix of the two.

Example: parameter_pass_2.py

```
# last parameter d is optional and defaulted to 0
def addNumbers(a, b, c, d=0):
    print(a, b, c, d)
    result = a + b + c + d
    return result

x = 2
y = 5
# positional parameter passing
addNumbers(1, x, 10, y)
addNumbers(1, x, 10)

# keyword parameter passing
addNumbers(a=1, d=y, c=10, b=x)
addNumbers(c=10, a=1, b=x)

# mixed ways of parameter passing
addNumbers(1, x, 10, d=y)
```

```
1 2 10 5
1 2 10 0
1 2 10 5
1 2 10 0
1 2 10 5
```

Notes:

- The parameter local variable **`d`** is set to 0 when the last parameter **`d`** was omitted.

> ❖ ***Advanced Usage of Parameters***
>
> Refer to the following webpage that discusses more advanced uses of keyword arguments.
> - Unlimited number of positional arguments.
> - Mandatory keyword parameters.
> - Passing a dictionary as arbitrary keyword arguments.
>
> https://treyhunner.com/2018/04/keyword-arguments-in-python/

1.4.3 Passing Mutable and Immutable Values

Recall that in Python all values (or objects) belong to one of the following two types:

- Immutable objects: the value cannot be modified.
- Mutable objects: the value can be modified.
- The following table shows which types of values (or variables) are mutable and immutable.

Mutable Types	***Immutable Types***
Lists, Sets, Dictionaries	Boolean, Integers, Floating-points, Strings, Tuples, FrozenSet

Passing a mutable value or object in a function call is different from passing an immutable value.

- Passing an immutable object or value is safe from modification by the function.
    - Any change made by the function remains localized in the function.
- Passing a mutable object or value is prone to modification by the function
    - For example, a list (which is of mutable type) passed to a function may have been modified when the function call returns.

Example 1: Passing Immutable Objects such as Integers

Consider the following function that receives an integer in the local variable number, and then multiply it by 2. The values before and after the multiplication are printed.

Example: parameter_pass_3.py

```
def functionA(number):
    print("before multiply", number)
    number = number * 2
    print("after multiply", number)

# test case 1
functionA(10)

# test case 2
x = 10
functionA(x)
print("after function call", x)
```

```
before multiply 10
after multiply 20
before multiply 10
after multiply 20
after function call 10
```

Notes:

- For test case 1, the parameter passed is the integer 10, which is an immutable object.
    - It is a constant and therefore it cannot be changed anyway.
- For test case 2, the parameter passed appears to be the variable **x**. It contains the integer 10, which is an immutable object.
    - In passing immutable object, the value 10 is copied to the parameter local variable.
    - Any change made to the parameter local variable changes only the parameter local variable.

      ```
      number = number * 2
      ```

    - The 10, which is an immutable value, cannot be modified.

Example 2: Passing Immutable Objects such as Strings and Tuples

Similarly, strings and tuples are immutable objects that are safe from modification by the called functions.

Example: parameter_pass_4.py

```
def functionB(name, dimension):
    # change name
    name = "Sony"
    # change dimension
    dimension = (1.5, 3, 2.4)

# create variables
product = "Apple"   # a string
dimension = (1.2, 2.5, 3.2)   # a tuple
# call function
functionB(product, dimension)
# after the function is called
print("product =", product)
print("dimension =", dimension)
```

```
product = Apple
dimension = (1.2, 2.5, 3.2)
```

Notes:

- The function call appears to have passed the variables `product` and `dimension` to the function, but this does not happen.
- The two variables contain a string and a tuple respectively.
    - Both are immutable.
    - As in the case of passing integers, these are copied to the parameter local variables of `functionB`.
- Any change made to the parameter local variable changes only the parameter local variable.

```
name = "Sony"
dimension = (1.5, 3, 2.4)
```

- The string and tuples, being immutable, are not modified.

Example 3: Passing Mutable Objects such as Lists

Passing mutable objects give the receiving functions an opportunity to modify these objects.

However, the following two types of changes are totally different:

- Assigning a new value to the parameter local variable.
- Calling a method or an operation on the list.

Consider the following examples. Assume that the function caller has passed a list.

Example #1: NO CHANGE TO THE LIST	*Example #2: CHANGED THE LIST*
`def functionC(alist):` `    alist = 10`	`def functionC(alist):` `    alist[0] = 10`
The statement assigns a new value to the parameter local variable `alist`. Any list that is passed to the function has not been modified. In fact, the reference to the list is lost. Without the reference, the list cannot be accessed.	The statement modifies the list. It assigns 10 to the element index 0 of the list that has been passed to the function. The parameter local variable `alist` has not been changed but the list content changed.
Example #3: CHANGED THE LIST	*Example #4: CHANGED THE LIST*
`def functionC(alist):` `    alist.clear()`	`def functionC(alist):` `    alist.remove(10)`
The statement calls a method on the list. The statement modifies the list.	The statement calls a method on the list. The statement modifies the list.

The following example show how a list is changed after passing it to a function.

Example: parameter_pass_5.py

```
def functionC(alist):
    count = len(alist)
    alist.reverse()
    return count

numlist = [1, 2, 3, 4, 5, 6, 7, 8]
result = functionC(numlist)
print("The list = ", numlist)
print("The return value = ", result)
```

```
The list = [8, 7, 6, 5, 4, 3, 2, 1]
The return value = 8
```

Notes:

- A list is created in the global scope. The variable `numlist` holds a reference to the list.
- The list is passed with a function call.
- In the function, the parameter local variable `alist` also holds a reference to the same list.
    - An function call to `id()` can reveal the reference.
- The statement calls a method on the list, which reverses the order of elements in the list.
    - This method call modifies the list.

❖ *Call by Value or Call by Reference*

Parameter passing is an important topic in programming language theory.

Call by value and call by reference represent two mainstream parameter passing methods.

- Call by value means that the value is passed to the function. Even if the caller submits a variable, actually its value is passed. The original variable cannot be modified.

```
number = 5
print(number)  # actually 5 is passed to the function, NOT the variable number
```

- Call by reference means that the reference of a variable is passed to the function. The function can use the reference to change the original variable.

Python, according to some experts, is neither call by value nor call by reference. It is actually both, or somewhere in between.

- In Python, all values, mutable and immutable, are objects.
- All variables hold the reference (or ID) of these objects.
- In parameter passing, actually the ID of these objects are copied to the function's parameter local variables.

Read more about this issue with the link below.

Reference: https://jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/

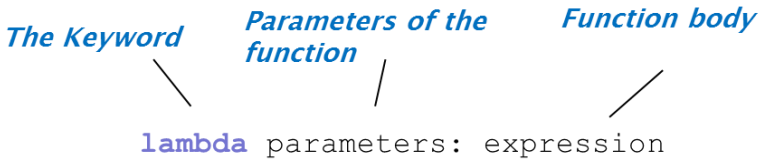
1.5 Lambda Functions

Lambda functions are functions defined without a name. They are also called anonymous functions.

- Normal functions are defined with the keyword **`def`**.
- Lambda functions are defined with the keyword **`lambda`**.

Lambda functions are simple functions that are needed for ad hoc tasks.

The structure of lambda function is simple.



- The keyword **`lambda`** begins the definition.
- The parameters are the parameter local variables of the function.
- The function body contains the instructions of the function. The result of the expression is the return value.

The Object Model of Functions

Lambda functions are objects like lists, dictionaries, integers, and strings. They can be assigned to a variable.

Example: lambda_func_1.py

```
twiceFunc = lambda num: num * 2

result = twiceFunc(10)
print(result)
```

```
20
```

Notes:

- The lambda function has one parameter named num. It returns twice the variable **`num`**.
- The function is assigned to the variable `twiceFunc`.
- The function call is made using the variable `twiceFunc`, which stores the reference to the lambda function.

The advantage of storing a function as an object is to store it in a variable, as well as passing it in a function call.

1.5.1 Example: Overall Course Score Finder with Replaceable Calculator

Consider the following example that calculates the overall course score (OCS) based on a list of 4 assignment scores and the examination score (OES). The mean assignment score (OCAS) and the examination score has ratio of 50:50.

There are at least two ways to calculate the OCAS:

- Average of 4 assignment scores.
- Average of best 3 out of 4 assignment scores.

The following implementation shows a function that accepts a replaceable assignment score calculator. The calculator is in the form of a lambda function.

Example: lambda_func_2.py

```
import statistics

# a function for calculating the overall course score
# based on assignment score list, exam score,
# and a function for finding mean assignment score
def overallScore(assignment, exam, calFunc):
    OCAS = calFunc(assignment)
    OES = exam
    return OCAS * 0.5 + OES * 0.5

# two different calculators
allCountFunc = lambda alist: statistics.mean(alist)
best3Of4Func = lambda alist: (sum(alist) - min(alist)) / 3

assignmentScore = [80, 60, 75, 90]
examScore = 70

# calculate using "All Assignment Counts"
OCS = overallScore(assignmentScore, examScore, allCountFunc)
print("Using 'All Assignment Counts' =", OCS)

# calculate using "Best 3 Out of 4"
OCS = overallScore(assignmentScore, examScore, best3Of4Func)
print("Using 'Best 3 Out of 4' =", OCS)
```

```
Using 'All Assignment Counts' = 73.125
Using 'Best 3 Out of 4' = 75.83333333333334
```

Notes:

- The parameter **`calFunc`** should receive a lambda function from the caller.
    - The function should provide a method to calculate OCAS from a list of 4 assignment scores.
- Two calculators are used in the program.
    - The first one (`allCountFunc`) calculates the mean of a list of numbers.
    - The second one (`best3Of4Func`) calculates the mean of best 3 out of 4, through subtracting the total sum with the lowest score.

```
allCountFunc = lambda alist: statistics.mean(alist)
best3Of4Func = lambda alist: (sum(alist) - min(alist)) / 3
```

- The function **`overallScore`** is called twice, each with a different assignment score calculator function object.
- An advantage for **`overallScore`** of accepting lambda function is extensibility.
    - A new way to calculate the overall assignment score can be implemented as a lambda function.
    - For example, a course calculates the overall assignment score as follows: 40% assignment 1 and 20% for the other 3 assignments.

```
anotherFunc = lambda alist: alist[0] * 0.4 + mean(alist[1:4]) * 0.6
```

    - The function **`overallScore`** needs no change. The caller passes `anotherFunc` to the function for changing the calculation method.

1.5.2 List Filtering with Lambda Functions

Lambda functions are often used with a number of built-in functions. The function `filter()` is used to filter a list based on the Boolean return value of a lambda function.

```
filter(func, list)
```

Notes:

- The first parameter is a function object.
    - The function should accept one parameter, which is one of the elements of the list (i.e. the second parameter).
- The second parameter is the original list to be filtered.
- This function returns a new filtered list.

Example 1: Integer List Filter

The following example shows how to use the filter function with a lambda function to filter an integer list.

Example: lambda_func_3.py

```
# EXAMPLE: integer list filtering
numlist = [6, 3, 4, 3, 2, 3, 6, 8, 3, 5, 1, 6, 4]

evenFilterFunc = lambda x: x % 2 == 0
filteredList = list(filter(evenFilterFunc, numlist))
print(filteredList)
```

```
[6, 4, 2, 6, 8, 6, 4]
```

Example 2: Phone Book Filter

The following example assumes to have a phone book, which is implemented as a list of dictionaries. Each dictionary contains two fields: **`name`** and **`phone`**.

Example: lambda_func_3.py

```
phoneBook = [{'name': 'Anders', 'phone': '68900000'},
             {'name': 'Betsy', 'phone': '80001111'},
             {'name': 'Chris', 'phone': '82223333'},
             {'name': 'Doris', 'phone': '60008888'},
             {'name': 'Eva', 'phone': '89102345'},
             {'name': 'Francis', 'phone': '62505050'}]


print(phoneBook)
print("Number of elements in list =", len(phoneBook))

filteredPhone = list(filter(lambda elem: elem['phone'][0] == '8', phoneBook))

print(filteredPhone)
print("Remaining elements in list =", len(filteredPhone))
```

```
[{'name': 'Anders', 'phone': '68900000'}, {'name': 'Betsy', 'phone': '80001111'},
{'name': 'Chris', 'phone': '82223333'}, {'name': 'Doris', 'phone': '60008888'},
{'name': 'Eva', 'phone': '89102345'}, {'name': 'Francis', 'phone': '62505050'}]
Number of elements in list = 6
[{'name': 'Betsy', 'phone': '80001111'}, {'name': 'Chris', 'phone': '82223333'},
{'name': 'Eva', 'phone': '89102345'}]
Remaining elements in list = 3
```

Notes:

- The lambda function returns **`True`** if the starting digit of phone number is 8.
- The filtered list has 3 elements remaining.
    - Out of the 6 original phone numbers, 3 of them start with 8.
- The filter is called with inline lambda function definition. The alternative is to define the lambda function with a variable, and then pass the variable to the filter.

```
start8Func = lambda elem: elem['phone'][0] == '8'
filteredPhone = list(filter(start8Func, phoneBook))
```

1.5.3 List Mapping with Lambda Functions

The function `map()` is used to map a list into another list. This is similar to list comprehension.

```
map(func, list)
```

Notes:

- This function works in a similar way as `filter()`.
- It returns a new mapped list, which is of the same length as the original list.

Example: Integer List Converter

The following example shows how to use the filter function with a lambda function to map the integers in a list to their square.

Example: lambda_func_4.py

```
# EXAMPLE: integer list mapping
numlist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

squareFunc = lambda x: x ** 2
newlist = list(map(squareFunc, numlist))
print(newlist)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

1.6 (Challenging) Runtime Stack in Functions

A professional programmer differs from a weak programmer in at least this aspect: knowing the details of programming. An important area to know is the mechanism of function calls.

This topic is related to the design of general programming languages. There are often a number of considerations such as performance, memory consumption, consistency, and platform independence. The discussion is not limited to Python.

1.6.1 The Runtime Stack

The runtime stack is a data structure where local variables are stored.

- The runtime stack is also called the stack memory.
  - Each program is allocated dedicated stack memory by the operating systems.
- When a function is called, its local variables are created on the stack.
- When the function call is finished, the local variables are deleted from the stack.
- This structure exists but hidden from the programmer.

Consider the function call in the following program. There is an integer passed to the function and there is a return value from the function.

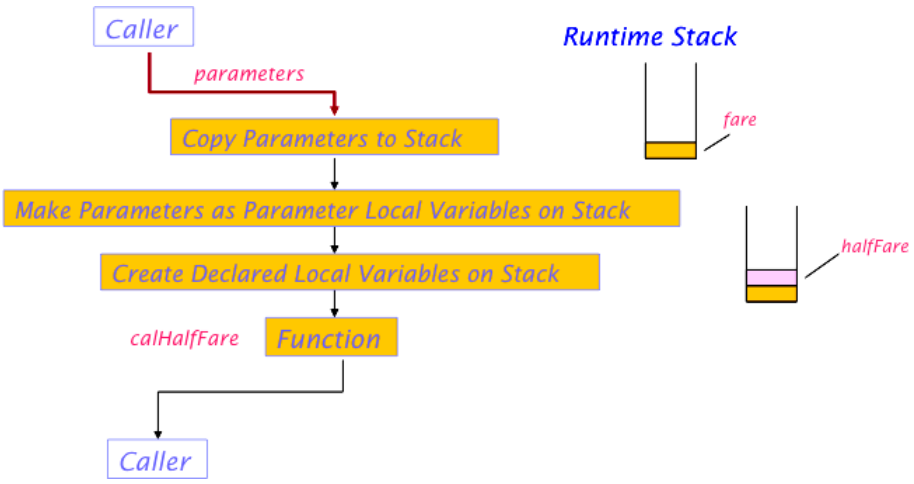
Example: runtime_stack.py

```
def calHalfFare(fare):
    halfFare = round(fare * 10 / 2) / 10
    return halfFare

result = calHalfFare(10.7)
print(result)
```

```
5.4
```

During Function Calls

The runtime stack is a place where local variables are created. The following figure illustrates the steps actually taken at a function call.

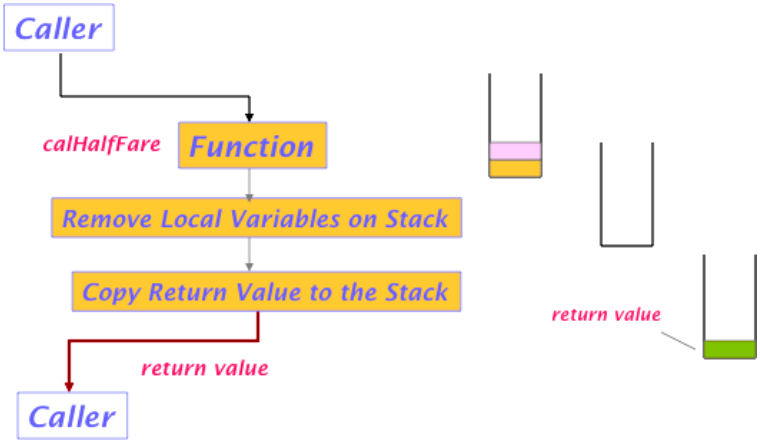


After the function is called:

- Copy the parameter to the stack
  - This step initializes the parameter local variable **`fare`**.
- Create other local variables as the function is being executed.
  - The local variable **`halfFare`** is created and added to the stack.

Function Call is Finishing

The following figure illustrates the steps taken when a function call is almost finished.

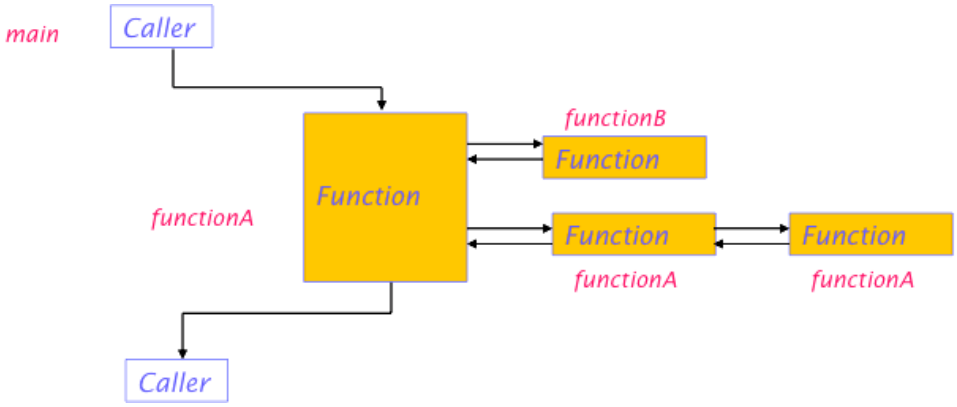


Notes:

- In this case, a value is to be returned back to the caller.
- At the **`return`** statement, the return value is left on the runtime stack.
- The function caller can then receive this return value.
    - The value is assigned to the variable **`result`**.

Multiple and Stacked Function Calls

The runtime stack can contain a lot of local variables if a function calls a function that calls functions and so on. Each function call is considered independent and has its own instances of local variables



Function Calls Take Additional Time

A function call takes additional time to complete because it involves some tasks to deal with the runtime stack.

- Copying of parameters to the stack
- Creation of local variables
- Removal of local variables from the stack
- Copying the return value

COMPS209F Data Structures, Algorithms and Problem Solving

Modular Programming and Software Development 2

Compare the time required to write a 10-line program and a 10000-line program. Logically, writing a 10000-line program requires 1000-fold effort and time than writing a 10-line program. However in reality, the effort is more than 1000 times. A very large task may add a level of complexity much more than the increase in the size.

For small programs, all code can be put inside a single module or file, as in most of the previous examples. However, having a single module with 10000 lines is not feasible for programming. For example, finding out errors in the program is difficult.

The added complexity requires new tools, processes, and methods to reduce time and effort.

For example, a library of 10 books is easy to store and to find the desired book. However, to manage a library of 10000 books, tools, processes, and methods are required:

- Better organization of books into categories (such as the Dewey system)
- Indexing for looking up.
- Recording system for check-in and checkout.

This chapter describes the modular programming approach for easing the difficulty in writing large programs.

2.1 Modular Programming

Everyone can write programs. Only expert programmers can develop large programs. Expert programmers know how to use methods and tools to deal with the added complexity of large programs.

2.1.1 Top-Down Approach and Divide-and-Conquer

The **top-down approach** breaks down a large problem into smaller problems. A problem will be repeatedly divided up until the problem is small enough to be handled comfortably.

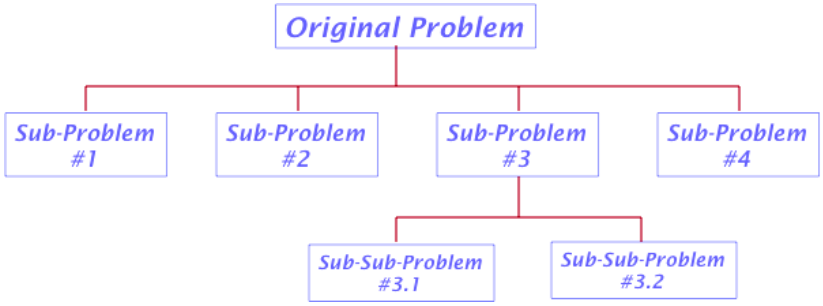
- Large problems are too complex to handle
- Small problems are easier to handle
- Therefore, large problems are broken down into small problems.

Expertise in software development is needed for the top-down approach to work.

- There are many ways to break down a large problem into smaller problems.
    - Good ways of breaking-down will make the small problems easier to solve.
    - Bad ways of breaking-down may create more problems or make small problems complex.
- Integration of solutions of small problem is also needed at the end.
    - Integration is sometimes not easy.

This method of solving problems is also called **divide-and-conquer**. A problem is divided up so that each smaller problem can be conquered. There are basically three steps in the top-down approach:

- Consider if divide and conquer is needed.
- Split up the problem until the smaller parts are in a form easier to handle.
- Integrate the smaller solutions to form the big solution.



2.1.2 Modular Programming

Modular programming is the way of software development that involves writing **modules** one by one, and then at the end integrates all these modules into a software system.

- A module is a part of a program.
- A module is designed to solve a small problem or perform a specific task.
- Each module is also known as a sub-routine, a sub-program, or a **component**.
- Each module contributes to a part of the functionality of the whole software system.

> ❖ ***Module or Component***
>
> The term module has a specific meaning in Python, which is not the same as the module in modular programming.
>
> - Python: The term module refers to a Python program file that can be imported into another program file. This module is a programming entity.
> - Modular Programming: The term module refers to a specific part of a program that is designed to solve a problem. This module is a conceptual entity.
>
> To avoid confusion, from this point onwards, the following rules are followed.
>
> - The term module refers to the Python's programming entity.
> - The term **component** refers to the module in modular programming.

A program is implemented by integrating a number of **components**. When the components are successfully integrated, they contribute to achieve the objectives of the whole system.

Modular programming also allows programmers to work more effectively.

- Work on each component one by one.
- Temporarily ignore the other components and the original problem

The characteristics of a suitable component are described in the following.

Characteristics	***Remarks***
Manageable size	• A component should be small enough so that its implementation should be straightforward.
One specific task	• A good component should be of **high cohesion**. • It means that the component should only perform one well-defined task. • If a component performs two or more distinctive tasks, then it should be divided into two separate modules.
Independent of other components	• Good components should also be of **low coupling**. • A component should, as far as possible, be logically independent of other components. • A component should minimize the complexity and extent of relationship with other components.

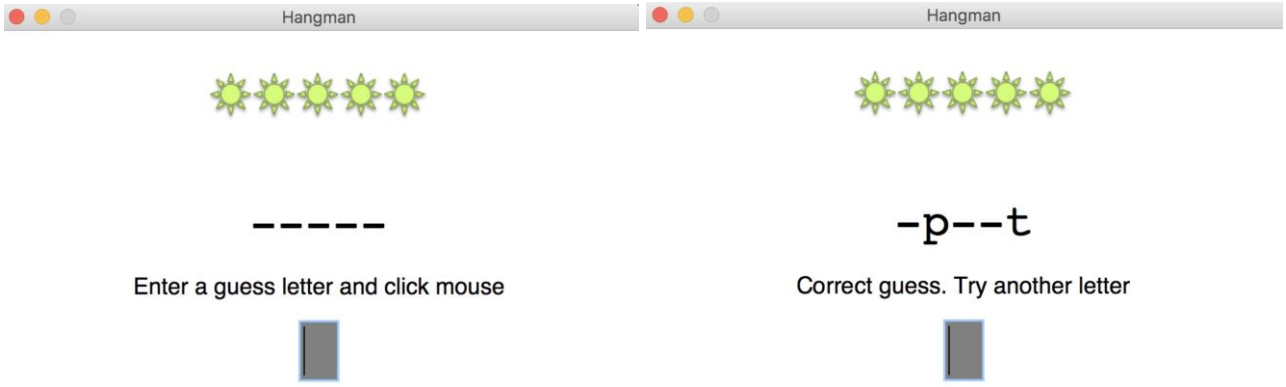
2.2 Example: Hangman

Hangman is a word spelling game. A word is randomly chosen from a dictionary. The game player then guesses a letter that the word may contain.

- If the word contains the letter, then the positions of the letters are shown.
- If the word does not contain the letter, then a live is lost.

The player has a certain number of lives. The game is over if all lives are lost.

The game should be running on graphical user interface (GUI) such as the one below.



Notes:

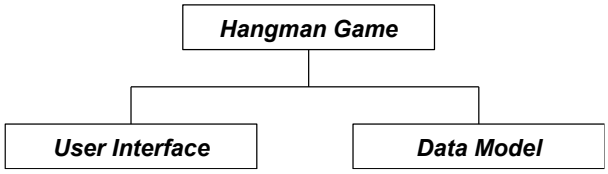
- The stars on the top indicate the number of lives.
    - The game should start with five lives. When the number of lives is zero, the game is over.
- The word to be revealed is displayed in the middle.
    - The dashes represent the hidden letters.
    - Correctly guessed letters are displayed at their respective positions of the game word.
- There is a message text box displaying messages to the player.
- The entry box allows the player to enter one character.
    - The player should click the mouse to confirm the input.

Top-Down: User Interface and Data Model

To apply the top-down approach, the original problem is broken down into sub-problems. The problem is the computation for the Hangman game.

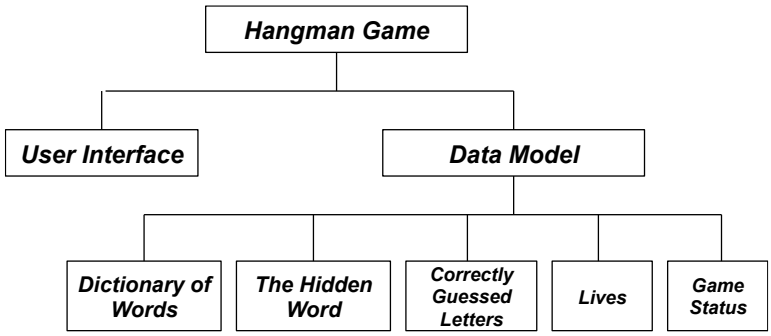
A common first step is to break down the game into the **user interface** sub-problem and the **data model** sub-problem.

- The user interface: includes the **view** (i.e. output) and the **controller** (i.e. input).
- The data model: includes the data involved in driving the game.



The data model stores information that enables the game to run.

- The dictionary of words (i.e. for randomly drawing the hidden word)
- The hidden word.
- The game word with letters correctly guessed.
- The number of remaining lives.
- The status of the game: ongoing, game over or game won.



Top-Down: Game Flow

The other target of top-down approach is to divide the game flow into stages or steps.

1. Setup the game (i.e. initialization).
2. Run the game.
3. Game end.

The setup-game part should consist of two steps:

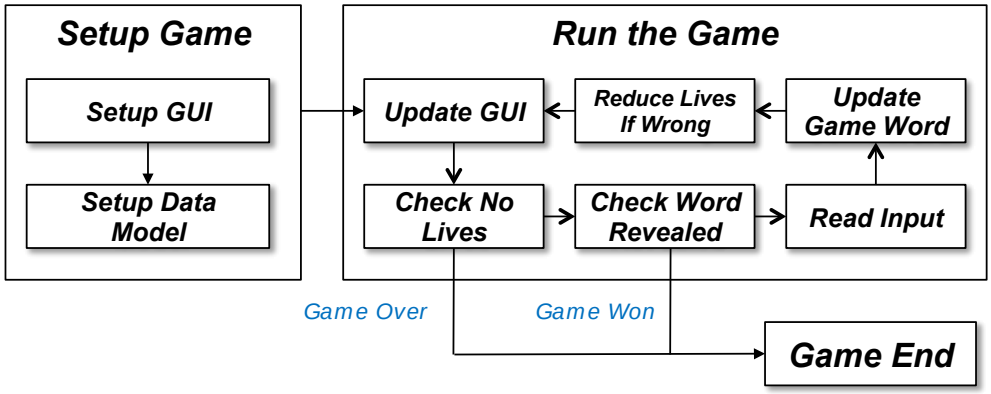
1. Setup the GUI
2. Setup the game's Data Model

The run-the-game part should be running in a loop. The game will continue reading guess letters from the user until either all lives are lost or the hidden word is revealed.

Repeat

1. Display game status on GUI (including the number of lives, the hidden word, etc).
2. If the whole word is revealed, go to Game End.
3. If there are no remaining lives, go to Game End.
4. Read input letter from the player.
5. Check whether the letter is found in the hidden word, and update the game word (i.e. correctly guessed letters filled in).
6. If the letter is not found, reduce the lives.

Through the above analysis, the problems of Setup Game and Run the Game are further divided up as follows.



2.2.1 Example: Hangman Setup GUI Design and Implementation

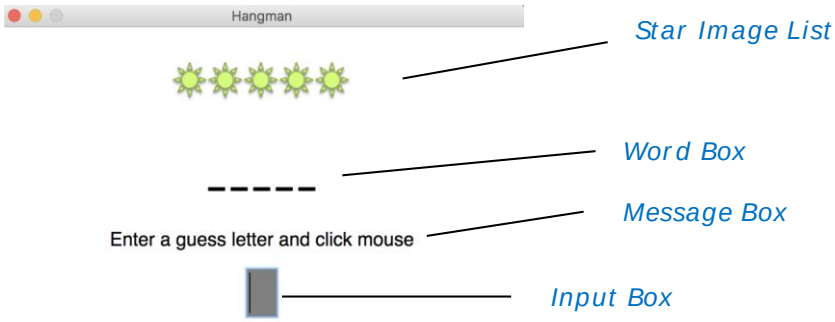
This section will focus on the design and implementation of the GUI part of Hangman.

- Design: a plan describing the components and their relation before they are programmed.
- Implementation: actually writing the components with a programming language.

Modular Design: Setup GUI

The next step is to design components based on the above design. Begin with Setup GUI.

Basically, each of the GUI components needs to be created. In addition, the program will use these components later, and therefore their references are stored in a data structure.



Component Name	***Task***
`createStarImage`	Create the object for the Star Image List
`createWordBox`	Create the object for the Word Box
`createMsgBox`	Create the object for the Message Box
`createInputBox`	Create the object for the Input Box

Finally, the above components are grouped together under the following component, which is designed to create the whole GUI.

Component Name	***Task***
`uiSetup`	Create all the GUI components

Recall that each component in modular programming approach should be of high cohesion and low coupling. The above design should be examined against the good practice.

- High cohesion: each component should do one task only.
    - Yes. Each component has one specific task.
- Low coupling: each component should be independent of each other.
    - Yes. The four components are independent of each other.
    - The `uiSetup` is designed to encapsulate or integrate the other four components.

Data-In and Data-Out Design: Setup GUI

The next step is to specify the data-in and data-out design of the components.

- Data-in: the information required by the component to do the task.
- Data-out: the information to be sent back to the caller by the component.

Component Name	***Data-In***	***Data-Out***
`createStarImage`	-	Object referring to the Star Image
`createWordBox`	-	Object referring to the Word Box
`createMsgBox`	-	Object referring to the Message Box
`createInputBox`	-	Object referring to the Input Box
`uiSetup`	-	A Composite Object referring to all the GUI components

Notes:

- These components have data-out only.
    - Their task is to create the GUI components.
    - They should return the GUI components, as object references.

Implementation of Setup GUI

Implementation means actually writing programs. At this point, the programming language to use must be decided. Python will be used for this example.

- Each component is implemented as a Python function.
- The data-in and data-out are implemented as parameters and return values.
- The **`graphics`** module (by John Zelle) is used as the GUI framework.
    - There are many GUI frameworks in Python but the graphics module is easy for beginner programmer.
    - It is not included in the Anaconda installation. Install the graphics module using the pip installation tool.

```
pip install --user http://bit.ly/csc161graphics
```

The following lists the implementation of the components as Python functions.

Example: hangman_2.py

```
def createWordBox():
    # create a text placeholder for the hidden word
    theWordBox = Text(Point(250, 150), "")
    theWordBox.setSize(36)
    theWordBox.setFace("courier")
    return theWordBox

def createMsgBox():
    # create a text placeholder for message
    theMsgBox = Text(Point(250, 100), "")
    theMsgBox.setSize(18)
    return theMsgBox

def createInputBox():
    # creste an input box
    theInputBox = Entry(Point(250, 50), 1)
    theInputBox.setSize(36)
    theInputBox.setFace("courier")
    return theInputBox

def createStarImage():
    # load and initialze star images for lives display
    starImageList = [None] * 6
    for i in range(1, 6):
        starImage = Image(Point(0, 0), "star" + str(i) + ".png")
        starImage.move(250, 250)
        starImageList[i] = starImage
    return starImageList
```

The functions' design is summarized in the following table.

Function	***Parameter***	***Return Value***
`createStarImage`	No	`list` object containing `Image`
`createWordBox`	No	`Text` object
`createMsgBox`	No	`Text` object
`createInputBox`	No	`Entry` object

Finally, the parent function `uiSetup` is listed below.

```
# setup all GUI components
def uiSetup():
    winWidth = 500  # create the window of size 500 x 300
    winHeight = 300
    win = GraphWin('Hangman', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight)
    theWordBox = createWordBox()
    theWordBox.draw(win)
    theMsgBox = createMsgBox()
    theMsgBox.draw(win)
    theInputBox = createInputBox().draw(win)
    starImageList = createStarImage()
    # store the GUI components in a dictionary
    ui = dict()
    ui['win'] = win
    ui['wordbox'] = theWordBox
    ui['msgbox'] = theMsgBox
    ui['inputbox'] = theInputBox
    ui['starimagelist'] = starImageList
    return ui
```

Notes:

- The function first calls **`GraphWin`** to create a GUI window using the **`graphics`** module.
- It then calls the 4 previous functions for creating the GUI components.
- It packages the object references of the GUI component in a dictionary.
- It returns the dictionary to the caller.

The Composite Object for GUI Component: UI

The dictionary **`ui`** is a composite object that packages all the GUI components into one object.

Dictionary Key	***Dictionary Value***
`win`	`GraphiWin` object
`starimagelist`	`list` object containing `Image`
`wordbox`	`Text` object for the Word Box
`msgbox`	`Text` object for the Message Box
`inputbox`	`Entry` object for input

Testing the Setup GUI Implementation

Implementation of components should always be tested. A test program is written for the purpose.

Example: hangman_testgui.py

```
import hangman_2

ui = hangman_2.uiSetup()

ui['wordbox'].setText("TEST")
ui['msgbox'].setText("A test message")
```

Notes:

- The test program first imports the functions that are stored in hangman_2.py.
- It then calls the function `uiSetup` and receives all the GUI components the return value as a dictionary.
- Finally, it sets text into the Word Box and Message Box using the functions provided by the `graphics` module.

2.2.2 Example: Hangman Update GUI Design and Implementation

This section describes components that make easier to use the GUI components.

Component Name	*Task*	*Data-In*	*Data-Out*
`updateMsg`	Set message to the Message Box	UI, Message	-
`updateGameWord`	Set updated Game Word	UI, Game Word	-
`updateStarImage`	Set Star Image according to remaining lives	UI, Remaining Lives	-
`waitGetInput`	Wait for the user input into the box and the confirm mouse click	UI	The input letter (lowercase)

Implementation of Update GUI

The following shows an implementation of the functions.

Example: hangman_2.py

```
def updateMsg(ui, text):
    ui['msgbox'].setText(text)

def updateGameWord(ui, theGameWord):
    ui['wordbox'].setText("".join(theGameWord))
```

```
def updateStarImage(ui, lives):
    # update the display lives
    for i in range(1, 6):
        ui['starimagelist'][i].undraw()
    if lives > 0:
        ui['starimagelist'][lives].draw(ui['win'])

# wait for player input to the inputbox
# return the guess letter after convert it into lowercase
def waitGetInput(ui):
    while True:
        ui['win'].getMouse()
        guess = ui['inputbox'].getText()
        ui['inputbox'].setText('')
        if guess.isalpha():
            guess = guess.lower() # convert it into lowercase
            break
        updateMsg(ui, MSG_2)
    return guess
```

Notes:

- These functions are designed to provide convenient way to update the GUI components.
- They hide away the **`graphics`** module.
    - Caller of these functions can update the GUI without knowing about the **`graphics`** module.
    - This is called **information hiding** or **encapsulation**.

Testing the Update GUI Functions

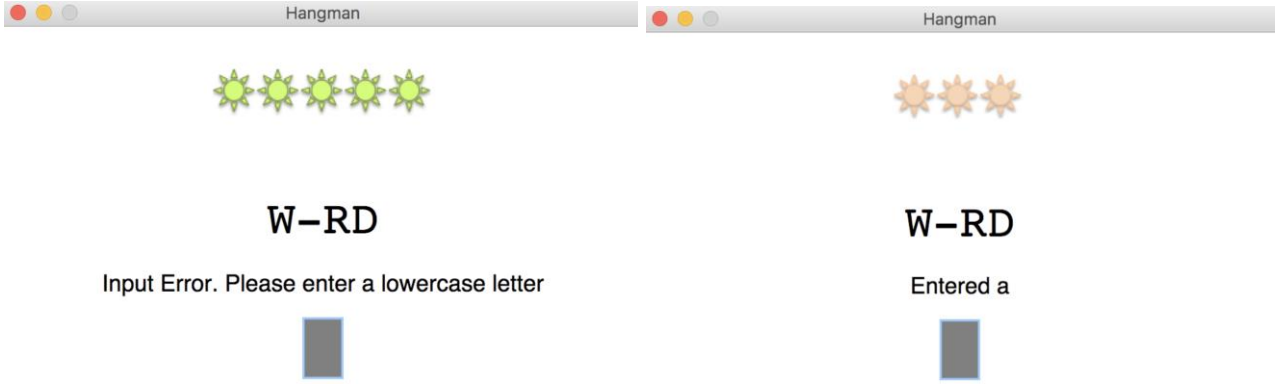
The following shows an implementation of testing program.

Example: hangman_testgui_2.py

```
import hangman_2

ui = hangman_2.uiSetup()

hangman_2.updateGameWord(ui, "W-RD")
hangman_2.updateMsg(ui, "From Testing Program - Click Mouse")
hangman_2.updateStarImage(ui, 5)

guess = hangman_2.waitGetInput(ui)

hangman_2.updateMsg(ui, "Entered " + guess)
hangman_2.updateStarImage(ui, 3)
```

Notes:

- The **`ui`** dictionary object received from `uiSetup`.
- The function call to **`hangman_2.waitGetInput(ui)`** is blocking – the program stops until the something has been entered into the Entry Box and confirmed by mouse click.



2.2.3 Example: Hangman Data Model

This section describes the Data Model components.

Recall that the Hangman game requires the following data items.

- The dictionary of words (i.e. for randomly drawing the hidden word)
- The hidden word.
- The game word with letters correctly guessed.
- The number of remaining lives.

Implementation of the Data Model: Data Container

The implementation of the Data Model include two aspects:

- The data container or data structure for holding the data
- The functions for easy handling of the data.

The following lists the Python data container for the Data Model.

Data Item	***Data Container***	***Remarks***
Word dictionary	A list of string	The list of all English words
The hidden word	A string	The randomly selected word
The game word with letters correctly guessed	A list of characters	Recording which letters have been guessed correctly
Number of remaining lives	An integer	

Implementation of the Data Model: Functions

The use of the Data Model can involve complex logic. Breaking it down into functions or components can simplify the complexity.

Component Name	***Task***	***Data-In***	***Data-Out***
`createDict`	Read words from a file and put them into a list	Filename	WordList: a list containing words
`pickWord`	Randomly choose a word from the dictionary	WordList	-
`checkGameWord`	Check if a guess letter is found in the game word. This component should update the game word	The Hidden Word The Game Word The Guess letter	True or False

The following shows an implementation of the 3 functions.

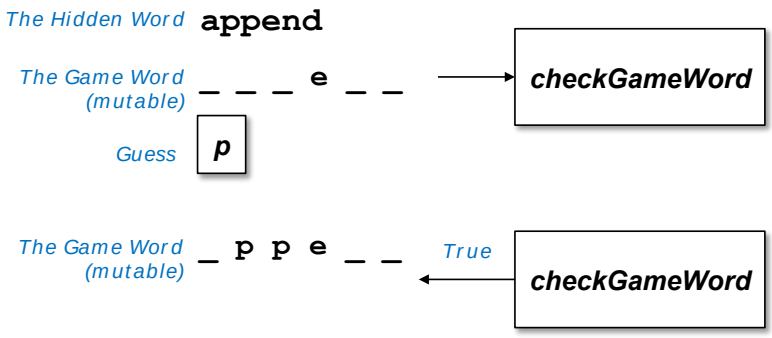
Example: hangman_2.py

```
# functions for dealing with the Data Model
def createDict(filename):
    # create dictionary
    wordList = list()
    with open(filename, "r") as infile:
        for line in infile:
            wordList.append(line.strip())
    return wordList
```

```
def pickWord(wordList):
    # randomly choose a word
    theHidden = random.choice(wordList)
    # create a list of the same length as the word
    # but contains dashes '-'
    theGameWord = ['-'] * len(theHidden)
    return (theHidden, theGameWord)

def checkGameWord(theHidden, theGameWord, guess):
    isFound = False
    index = 0
    for ch in theHidden:
        if ch == guess:
            isFound = True
            theGameWord[index] = guess
        index += 1
    return isFound
```

Notes about the function **`checkGameWord`**:

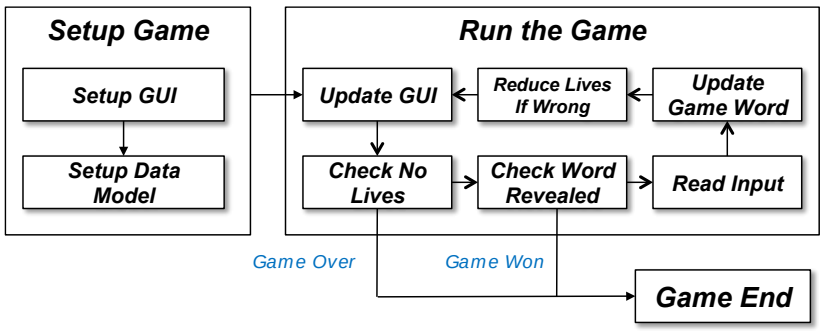
- The function takes in the original hidden word, the game word (which records which letters guessed correctly, and the guess letter).
- It then checks if the guessed letter is found the word.
- It updates the Game Word, which is a mutable list and can be changed.
- If the letter is found, the function returns **`True`**. If otherwise, return **`False`**.



2.2.4 Example: Hangman Integration

All the supporting components have been implemented. The final step is to **integrate** these components and get them working together. This integration is realized in the main program part.

Recall the following design of the running of the game.



The following shows an implementation of the above design.

Example: hangman_2.py

```
# the main program
if __name__ == "__main__":

    # game setup
    ui = uiSetup()
    lives = 5  # initialize number of lives
    wordList = createDict('dict.txt')
    theHidden, theGameWord = pickWord(wordList)
    # print(theHidden)
    updateMsg(ui, MSG_1)

    # the game loop
    while True:
        updateGameWord(ui, theGameWord)
        updateStarImage(ui, lives)
        # check if all hidden chars are revealed
        if theGameWord.count('-') == 0:
            updateMsg(ui, MSG_6)
            break
        # check if all lives are lost
        if lives <= 0:
            updateMsg(ui, MSG_5)
            break
        # read letter from input box
        guess = waitGetInput(ui)
        # check if the hidden word contains the guess letter
        isFound = checkGameWord(theHidden, theGameWord, guess)
        # if the guess letter not found, subtract lives
        if not isFound:
            updateMsg(ui, MSG_3)
            lives -= 1  # reduce the number of lives
        else:
            updateMsg(ui, MSG_4)
```

Notes:

- The five key variables are initialized in the game setup part.

Variable Names	*Type*	*Remarks*
`ui`	Dictionary	Contains all the GUI components
`lives`	Integer	Remaining number of lives
`wordlist`	List of Strings	List of words loaded from a word file
`theHidden`	String	The randomly chosen word
`theGameWord`	List of Characters	The game word

- The game loop is a while loop structure. It allows the repeatedly updating of the game.
    - It is a turn-based game. The loop iterates once per turn.
- The text messages are defined in the program separately.

Example: hangman_2.py

```
# game text messages
MSG_1 = "Enter a guess letter and click mouse"
MSG_2 = "Input Error. Please enter a lowercase letter"
MSG_3 = "Wrong guess. Try another letter"
MSG_4 = "Correct guess. Try another letter"
MSG_5 = "Game Over"
MSG_6 = "Success! Well done!"
```

COMPS209F Data Structures, Algorithms and Problem Solving

Abstraction and Software Development 3

Programs are written to solve problems. Real world problems are often very complex and large, and they challenges programmers to be able to write programs of increasing complexity and size.

An effective way to deal with complexity is **abstraction**. Abstraction means creating a simplified illusion of the reality. Abstraction is a means of simplification.

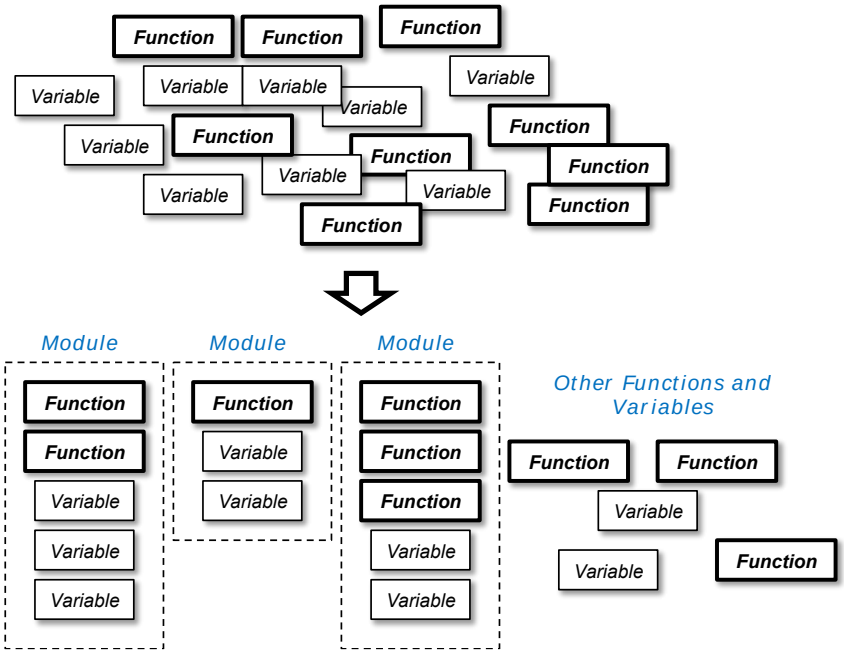
In computer programming, simplification can be achieved by **information hiding**. The information that is less relevant is hidden away. Programmers can focus on the more important information. Their jobs are made easier.

3.1 Python Classes

Large programs have many variables and computation code. The modular programming approach would facilitate computation code to be broken down into modules. However, there are still a lot of variables and functions in large problems. The complexity is reduced but not enough.

Fortunately, the following relations are commonly observed.

- Some variables are designed for the same task, component, or problem.
- Some variables and functions are designed for the same task, component, or problem.
- Most other variables and functions are unrelated.

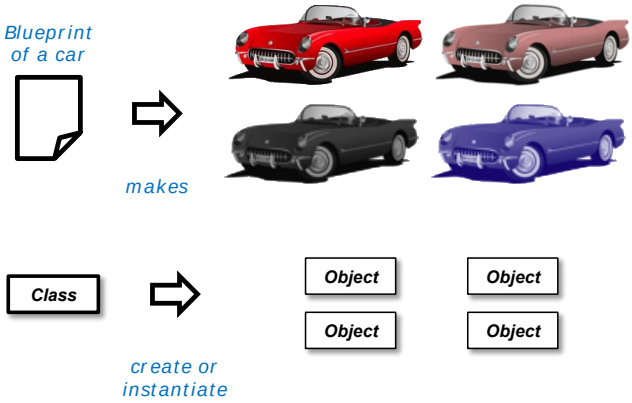


The complexity can be reduced with grouping related variables and functions together.

- The grouped together functions and variables form a module or a component.
- This is a more advanced concept of module or component.
- The original module is a function only, and a better form of module should include related variables as well.

Python offers a structure called classes that allow programmers to easily group together related variables and functions.

- A class is a template for creating objects.
- A class is neither a data container nor a variable. It cannot store data.
- An object is created out of a class. It has the variables and functions that are as defined in the class. It can store data.



A class is like a blueprint of a car. It is not a car, but it contains the design of a car. With the blueprint, many cars can be created.

3.1.1 Example: The Python List Class

The list has been used heavily in examples of this and the previous course. List is a standard Python class.

The actual lists are objects of the type **`list`**. The following shows examples of creating list objects.

```
alist = list()  # create a list object

blist = [1, 2, 3, 4]  # create a list object with initialized values

list()  # create a list object but the reference is lost
```

Notes:

- The variables `alist` and `blist` contains reference to different list objects.
- The creation of a list object is identified by a reference. If the reference is not stored in a variable, then the program can no longer compute with the list object.

List objects can be destroyed with the del keyword.

```
alist = list()  # create a list object

del alist  # delete the list object referenced by the variable alist
```

List objects can store data. List objects can be used in computation through methods of the **`list`** class.

3.1.2 Python New Class Definition

The following shows how to define a Python class.

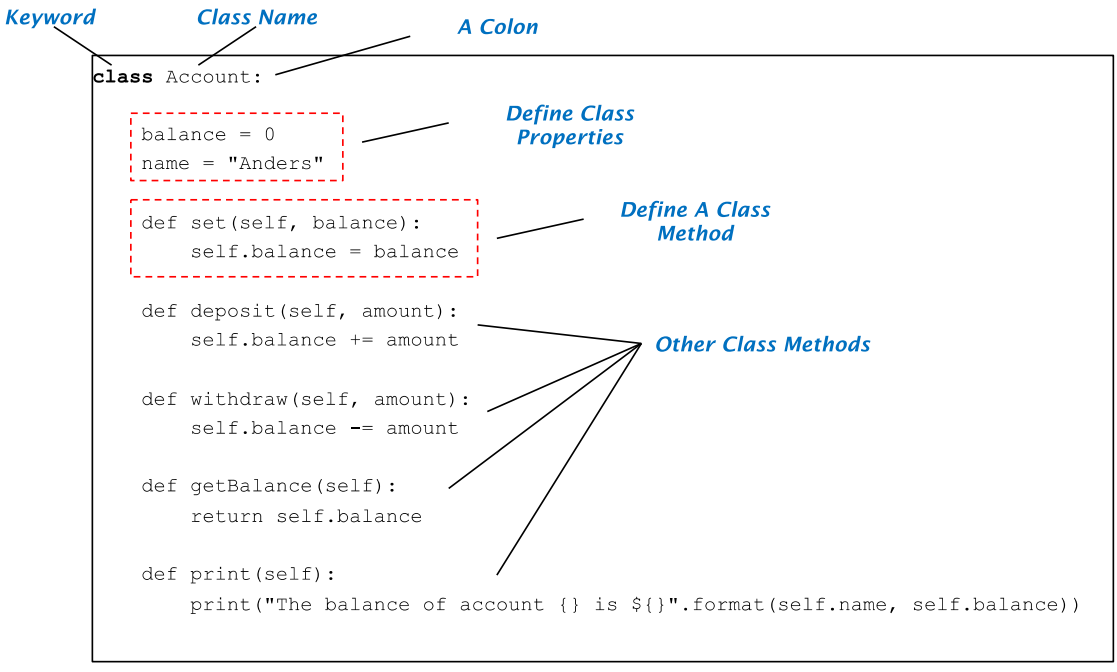
Example: account.py

```
class Account:
    balance = 0
    name = "Anders"

    def set(self, balance):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def getBalance(self):
        return self.balance

    def print(self):
        print("The balance of account {} is ${}".format(self.name, self.balance))
```

Definition of a Class

The definition of a class contains the following parts.

- The keyword `class`.
- The name of the class. In this case the name is `Account`.
- The class properties.
    - Class properties will become actual variable in objects.
- The class methods.
    - Class methods are functions that belong to objects.



Create Objects of a Class

The following shows an example of using the class **`Account`**.

- Create object of the class.
- Call methods of the object.

Example: account.py

```
if __name__ == "__main__":

    acct1 = Account()    # create an object of the class Account

    acct1.print()        # call the method print on the object
    acct1.set(100)       # call the method set on the object
    acct1.print()
```

```
The balance of account Anders is $0
The balance of account Anders is $100
```

Refer to the Class Properties and Methods

Object references (i.e. variables holding the objects) must be included when referring to the objects' methods or properties.

- The class properties and methods have their names.
- To refer to them in programs, however, using the name alone is not enough.
- The dot operator is used to combine the object reference and a method.

  `<object>.<method>(<parameters>)`
- Similarly, the dot operator is used to combine the object reference and a property.

  `<object>.property`

The following examples illustrate how to use acct1 and acct2, which holds reference to two Account objects, to refer to its methods and properties.

Example: account_test.py

```
from account import Account

acct1 = Account()
acct2 = Account()

acct1.set(100)
acct1.print()

acct2.balance = 200
acct2.name = "Betsy"
acct2.print()
```

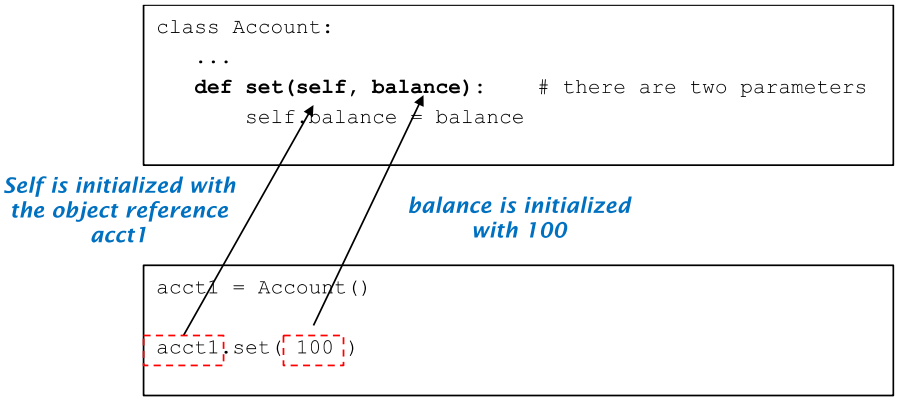
```
The balance of account Anders is $100
The balance of account Betsy is $200
```

The Self Parameter

There is an apparent inconsistency in the number of parameters in the method definition and the method call.

Example

```
class Account:
   ...
    def set(self, balance):     # there are two parameters
        self.balance = balance

    acct1 = Account()     # create an object of the class Account

    acct1.set(100)        # only one parameter between the brackets.
```



Notes:

- The name `self` for parameter local variable is special.
- It is always initialized with the object reference (i.e. variable) with which this method call is made.
- In the above example, `self` is initialized with the same object reference as `acct1`. Therefore the following two will do the same computation.

```
self.balance = 100
```

```
acct1.balance = 100
```

The Init Method

Python class definition allows the definition of a `__init__` method. This method is executed right after the creation of an object of the class.

- Useful for initialization of class properties through parameters.
- Creation of object and initialization can be done together.

The following example shows adding a `__init__` method to allow the setting of account name at object creation.

Example: account_2.py

```
class Account:

    def __init__(self, name):
        self.balance = 0
        self.name = name

    # other methods are omitted here to reduce space
...

if __name__ == "__main__":
    acct1 = Account("Anders")
    acct1.deposit(5000)
    acct1.print()

    acct2 = Account("Betsy")
    acct2.deposit(1200)
    acct2.withdraw(500)
    acct2.print()
```

```
The balance of account Anders is $5000
The balance of account Betsy is $700
```

Notes:

- The advantage of using `__init__` is to allow property initialization with parameters.
    - In the above example, the account name is provided with Account object creation.
- Normally, property initialization is done in this method.
- The `__init__` function is often called **constructor** in object-oriented languages such as Java.

3.2 Abstract Data Types (ADT)

Abstract data type (ADT) is a method of simplification of programming complexity.

- The abstraction includes both the operations (i.e. the functions) and the data representation (i.e. the variables or properties).
- It is a more advanced type of abstraction than that in modular programming.
- A data model (i.e. such as a bank account) does not offer much without the operations (i.e. deposit, withdraw, set, etc).

Data model is an essential part of any non-trivial programs. New data types are needed for implementation of new data models. Keep in mind that a useful data type needs both data and operations.

The following shows some examples of data models. The principle of ADT requires the definition of both data and operations for these data models.

Data Model	***Data***	***Operations***
Bank Account	Account Name Balance	Deposit, Withdraw, Set, Print
Assignment Record	Student Unique Identity (OUID) Name Scores of Assignments	Set Score of an Assignment Get Score of an Assignment Total and Average of all Assignments Test if Student Has Passed

The Python class provides a convenience tool for implementing ADT for new data types.

3.2.1 Example: An ADT for Rational Numbers

A rational number is a fraction made up of two integers — A and B as integers in a ratio, A/B, provided that B is not zero.

- For examples, $\frac{1}{2}$ and $\frac{3}{4}$ are rational numbers.
- This data type provides an exact representation of fractional values.
- It is not affected by the intrinsic precision limitation of the floating-point representation.

Like integers and floating-point types, rational numbers have similar basic operations including addition, subtraction, multiplication, and division.

$a / b + c/ d = (a \times d + b \times c) / (b \times d)$ *(definition A: addition)*

$a / b * c / d = (a \times c) / (b \times d)$ *(definition B: multiplication)*

Division and subtraction can be defined as:

$(a / b) / (c / d) = (a \times d) / (b \times c)$ *(definition C: division)*

$a / b – c / d = (a \times d - b \times c) / (b \times d)$ *(definition D: subtraction)*

There is a simplifying operation: the rational number 2/4 should be the same as 1/2:

$a / b \sim c / d$ *if and only if* $a \times d = b \times c$ *(definition E)*

The Data Model Part

Each rational number has two integers, the numerator and the denominator. A Python class is defined that includes two integers as the class properties.

Example: rational.py

```
# Rational ADT

class Rational:
    def __init__(self, numerator=0, denominator=0):
        self.numerator = numerator
        self.denominator = denominator

rat1 = Rational()      # create the rational number 0
rat1 = Rational(1, 2)  # create the rational number 1/2
```

Notes:

- The `__init__` function initializes (i.e. defines) two properties: **`numerator`** and **`denominator`**.
- The parameters are optional. If they are not provided, then the two properties are initialized to 0.

The Operation Part

The operation part of the Rational ADT is implemented as class methods.

Example: rational.py

```
class Rational:

    def __init__(self, numerator=0, denominator=0):
        self.numerator = numerator
        self.denominator = denominator

    def print(self):
        print("{} / {}".format(self.numerator, self.denominator))

    def add(self, another):  # another is another Rational object
        self.numerator = self.numerator * another.denominator + another.numerator *
self.denominator
        self.denominator = self.denominator * another.denominator

    def sub(self, another):  # another is another Rational object
        self.numerator = self.numerator * another.denominator - another.numerator *
self.denominator
        self.denominator = self.denominator * another.denominator

    def muliply(self, another):  # another is another Rational object
        self.numerator = self.numerator * another.numerator
        self.denominator = self.denominator * another.denominator

    def divide(self, another):  # another is another Rational object
        self.numerator = self.numerator * another.denominator
        self.denominator = self.denominator * another.numerator
```

```
rat1 = Rational(3, 4)  # create the rational number 3/4
rat2 = Rational(1, 2)  # create the rational number 1/2

rat1.print()
rat2.print()

rat1.add(rat2)
rat1.print()

rat3 = Rational(3, 2)
rat3.multiply(rat1)
rat3.print()
```

```
3 / 4
1 / 2
3 / 8
9 / 16
```

3.2.2 Example: An ADT for the Word List in Hangman

The Word List in the Hangman program provides the hidden word from a word list. The word list is obtained from a file containing many words.

The Data Model Part

The Word List holds a list of words. The data model part consists of only a list, which is used to store strings of words read from a file.

Example: wordlist.py

```
class WordList:

    def __init__(self, filename):
        # create dictionary
        self.wordList = list()
        with open(filename, "r") as infile:
            for line in infile:
                self.wordList.append(line.strip())

# the main program
if __name__ == "__main__":

    wordlist = WordList("dict.txt")
```

The Operation Part

The operation part has only one method **`pickWord`**.

Example: rational.py

```
import time, random, math, string

class WordList:

    def __init__(self, filename):
        # create dictionary
        self.wordList = list()
        with open(filename, "r") as infile:
            for line in infile:
                self.wordList.append(line.strip())

    def pickWord(self):
        # randomly choose a word
        theHidden = random.choice(self.wordList)
        # create a list of the same length as the word
        # but contains dashes '-'
        theGameWord = ['-'] * len(theHidden)
        return (theHidden, theGameWord)

# the main program
if __name__ == "__main__":
    wordlist = WordList("dict.txt")

    print(wordlist.pickWord())  # pick a random word
    print(wordlist.pickWord())  # pick a random word
    print(wordlist.pickWord())  # pick a random word
```

3.2.3 Example: An ADT for the GUI Components in Hangman

Defining an ADT for the GUI components in the Hangman program can further reduce the complexity.

All the related functions and the GUI component object references are grouped together under the same ADT or class.

Example: hangman_gui.py

```
from graphics import *
import time, random, math, string

class GUI:
    # setup all GUI components
    def __init__(self):
        self.winWidth = 500  # create the window of size 500 x 300
        self.winHeight = 300
        self.win = GraphWin('Hangman', self.winWidth, self.winHeight)
        self.win.setCoords(0, 0, self.winWidth, self.winHeight)
        self.theWordBox = self.createWordBox()
        self.theWordBox.draw(self.win)
        self.theMsgBox = self.createMsgBox()
        self.theMsgBox.draw(self.win)
        self.theInputBox = self.createInputBox().draw(self.win)
        self.starImageList = self.createStarImage()

    def createWordBox(self):
        # create a text placeholder for the hidden word
        theWordBox = Text(Point(250, 150), "")
        theWordBox.setSize(36)
        theWordBox.setFace("courier")
        return theWordBox

    def createMsgBox(self):
        # create a text placeholder for message
        theMsgBox = Text(Point(250, 100), "")
        theMsgBox.setSize(18)
        return theMsgBox

    def createInputBox(self):
        # creste an input box
        theInputBox = Entry(Point(250, 50), 1)
        theInputBox.setSize(36)
        theInputBox.setFace("courier")
        return theInputBox

    def createStarImage(self):
        # load and initialze star images for lives display
        starImageList = [None] * 6
        for i in range(1, 6):
            starImage = Image(Point(0, 0), "star" + str(i) + ".png")
            starImage.move(250, 250)
            starImageList[i] = starImage
        return starImageList

    def updateMsg(self, text):
        self.theMsgBox.setText(text)

    def updateGameWord(self, theGameWord):
        self.theWordBox.setText("".join(theGameWord))
```

```
    def updateStarImage(self, lives):
        # update the display lives
        for i in range(1, 6):
            self.starImageList[i].undraw()
        if lives > 0:
            self.starImageList[lives].draw(self.win)

    # wait for player input to the inputbox
    # return the guess letter after convert it into lowercase
    def waitGetInput(self):
        while True:
            self.win.getMouse()
            guess = self.theInputBox.getText()
            self.theInputBox.setText('')
            if guess.isalpha():
                guess = guess.lower() # convert it into lowercase
                break
        return guess

if __name__ == "__main__":
    theGUI = GUI()
    theGUI.updateMsg("Test the new class")
    theGUI.updateGameWord("CLASS")
    theGUI.updateStarImage(5)
```

Notes:

- The **`self`** parameter is used to refer to all class properties and methods.
- The role of the original UI dictionary is replaced by the class properties.
    - All GUI components are hidden inside the class definition.

3.2.4 Example: Using the Defined ADTs in Hangman

The re-developed Hangman program has three different Python files.

- The file wordlist.py contains the Word List ADT definition.
- The file hangman_gui.py contains the GUI ADT definition.
- The file hangman_3.py contains the main program.

Example: hangman_3.py

```
from wordlist import WordList
from hangman_gui import GUI

...
def checkGameWord(theHidden, theGameWord, guess):
    isFound = False
    index = 0
    for ch in theHidden:
        if ch == guess:
            isFound = True
            theGameWord[index] = guess
        index += 1
    return isFound

# the main program
if __name__ == "__main__":
    # game setup
    # create objects of the ADTs
```

```
theGUI = GUI()
theWordList = WordList('dict.txt')

lives = 5  # initialize number of lives
theHidden, theGameWord = theWordList.pickWord()
print(theHidden)
theGUI.updateMsg(MSG_1)

# the game loop
while True:
    theGUI.updateGameWord(theGameWord)
    theGUI.updateStarImage(lives)
    # check if all hidden chars are revealed
    if theGameWord.count('-') == 0:
        theGUI.updateMsg(MSG_6)
        break
    # check if all lives are lost
    if lives <= 0:
        theGUI.updateMsg(MSG_5)
        break
    # read letter from input box
    guess = theGUI.waitGetInput()
    # check if the hidden word contains the guess letter
    isFound = checkGameWord(theHidden, theGameWord, guess)
    # if the guess letter not found, subtract lives
    if not isFound:
        theGUI.updateMsg(MSG_3)
        lives -= 1  # reduce the number of lives
    else:
        theGUI.updateMsg(MSG_4)

theGUI.updateGameWord(theGameWord)
theGUI.updateStarImage(lives)
```

Notes:

- The first two import statements include the definition of the Word List and the GUI ADT into this program.
    - The name `WordList` is referring to the `WordList` defined in `wordlist.py`.
    - The name `GUI` is referring to the `GUI` defined in `hangman_gui.py`.
- This main program needs no reference the `graphics` module.
    - The graphics module is encapsulated inside the GUI ADT.
    - The hidden away of the graphics module removed the need to handle the GUI components in the main program.

3.3 The Python String Classes

String is one of the most used object class in Python. Strings look like a list of characters but they are immutable. Strings do not support item assignment with the square bracket (`[]`) operator.

```
s = "Python programming"
s[0] = 'A'  # ERROR: No Item Assignment
```

There are normally two ways to edit the content of strings.

- Convert a string into a list, which is mutable. Edit the list, and finally convert it back to a string.
- Use the methods in the string class.

3.3.1 String Editing via Converting to a List

Example: String Manipulation via a List

The following shows the whole program of replacing spaces in a string with underscore characters.

Example: string_2.py

```
s = "Python programming"

# convert the string to a list
strlist = list(s)
print(strlist)

# iterate through the list
index = 0
for index in range(index, len(strlist)):
    if strlist[index] == ' ':
        strlist[index] = '_';  # replace space with underscore

# convert it back to string
s = "".join(strlist)
print(s)
```

```
['P', 'y', 't', 'h', 'o', 'n', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n',
'g']
Python_programming
```

Convert String into a List

Use the built-in function `list()` to create a list out of a string.

```
s = "Python programming"

strlist = list(s)  # convert the string into a list
print(strlist)
```

Iterate the Characters in the List

The list created from a string will contains a sequence of characters. The usual list traversal method can be used to process the character.

The following shows a loop that replaces any space character with the underscore character in the list.

```
# iterate through the list
index = 0
for index in range(index, len(strlist)):
    if strlist[index] == ' ':
        strlist[index] = '_';  # replace space with underscore
```

Convert a List of Characters to a String

Finally the list of characters can be converted back to a string using a very Python-like method.

```
# convert it back to string
s = "".join(strlist)
print(s)
```

3.3.2 String Editing with String Methods

Strings are Python objects and the class provides a lot of methods. It is likely that one of the methods can do the task.

For example, there is a method called **`replace()`**, that can replace all occurences of a given character with another character.

Example: string_3.py

```
s = "Python programming"

# replace all space characters with underscore characters
snew = s.replace(' ', '_')
print(snew)
```

```
Python_programming
```

Notes:

- Strings are immutable.
- For methods that seem to change a string, in fact, the original string is not changed.
    - The variable `s` still holds the original string.
    - The new string is returned in the method call, and stored in variable `snew`.

3.3.3 The String Methods

Students are expected to research the methods that are provided by the string class. The following shows a selected list of methods. For the details, please refer to online references.

Methods of string objects	
`capitalize()`	Converts the first character to upper case. Returns a new string.
`casefold()`	Converts string into lower case. Returns a new string.
`center()`	Returns a centered string
`count(value[,start,end])`	Returns the number of times a specified value occurs in a string
`encode(encoding='utf-8')`	Returns an encoded version of the string
`endswith(value[,start,end])`	Returns true if the string ends with the specified value
`expandtabs(tabsize=8)`	Sets the tab size of the string
`find(value[,start,end])`	Searches the string for a specified value and returns the position of where it was found
`format(...)`	Formats specified values in a string
`format_map(mapping)`	Formats specified values in a string
`index(value[,start,end])`	Searches the string for a specified value and returns the position of where it was found
`isalnum()`	Returns True if all characters in the string are alphanumeric
`isalpha()`	Returns True if all characters in the string are in the alphabet
`isdecimal()`	Returns True if all characters in the string are decimals
`isdigit()`	Returns True if all characters in the string are digits
`isidentifier()`	Returns True if the string is an identifier
`islower()`	Returns True if all characters in the string are lower case

`isnumeric()`	Returns True if all characters in the string are numeric
`isprintable()`	Returns True if all characters in the string are printable
`isspace()`	Returns True if all characters in the string are whitespaces
`istitle()`	Returns True if the string follows the rules of a title
`isupper()`	Returns True if all characters in the string are upper case
`join(iterable)`	Joins the elements of an `iterable` to the end of the string
`ljust(width[,fillchar])`	Returns a left justified version of the string
`lower()`	Converts a string into lower case
`lstrip([charset])`	Returns a left trim version of the string, and charset are the characters to trimmed
`maketrans()`	Returns a translation table to be used in translations
`partition(sep)`	Returns a tuple where the string is parted into three part using the separator
`replace(old, new[,count])`	Returns a string where a specified value is replaced with a specified value
`rfind(value[,start,end])`	Searches the string for a specified value and returns the last position of where it was found
`rindex(value[,start,end])`	Searches the string for a specified value and returns the last position of where it was found
`rjust(width[,fillchar])`	Returns a right justified version of the string
`rpartition(sep)`	Returns a tuple where the string is parted into three parts
`rsplit(sep=None, maxsplit=-1)`	Splits the string at the specified separator, and returns a list
`rstrip([charset])`	Returns a right trim version of the string, and charset are the characters to trimmed
`split(sep=None, maxsplit=-1)`	Splits the string at the specified separator, and returns a list
`splitlines()`	Splits the string at line breaks and returns a list
`startswith(value[,start,end])`	Returns true if the string starts with the specified value
`strip([charset])`	Returns a left and right trim version of the string, and charset are the characters to trimmed
`swapcase()`	Swaps cases, lower case becomes upper case and vice versa
`title()`	Converts the first character of each word to upper case
`translate(table)`	Returns a translated string
`upper()`	Converts a string into upper case
`zfill(width)`	Fills the string with a specified number of 0 values at the beginning

3.4 Software Development Methodologies

Software development is a process of developing a software entity for some specific applications or general usages.

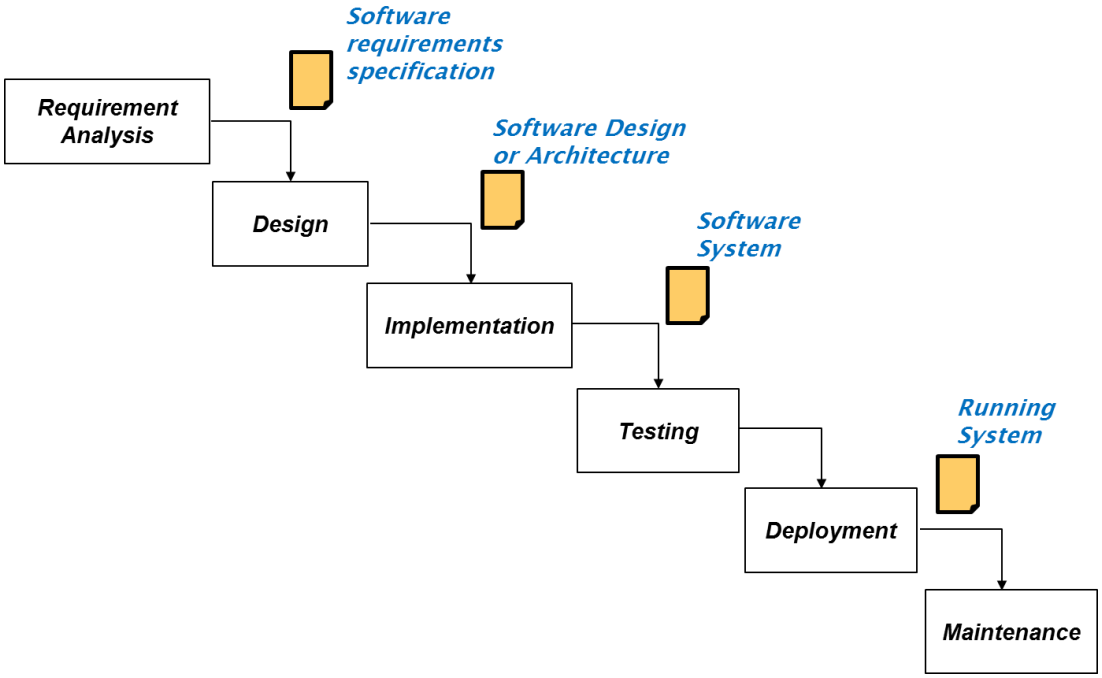
- Programming is only one aspect of software development.
- The process involves many tasks before and after programming, such as deciding requirements (functional and non-functional), design, testing, and maintenance.

- Large software requires project management
    - Planning and scheduling.
    - Involving more than one programmer
    - Involving other people such as project manager, specialists in database, user interface design, testing, etc.
- The objectives include both functional and non-functional requirements such as speed, efficiency, screen size (for mobile phones), etc.

Software development can be very challenging. The study of **software engineering** is a discipline that considers engineering approaches to software development. It is about the application of scientific and technological knowledge for the development of software that will more likely meet requirements including time and cost.

3.4.1 The Water Fall Model of Software Development

The Waterfall Model was believed to be the first widely-used software development methodology. It considers software development to involve the following stages.



- Requirement and specification.
    - The requirements of the software system include functional requirements and non-functional requirements.
    - Functional requirements are related to functionality.
    - Non-functional requirements are about characteristics of the software system such as performance and capacity.
    - These requirements are gathered and analysed.
    - The specification of the software system is a documentation of these considerations and analysis.

- Design.
    - The design is how the requirements can be satisfied through logical and mathematical computation.
    - It employs techniques and tools to manage complexity and size of problems.
    - For example, modular programming for breaking down a large problem into smaller sub-problems.
    - Other techniques such as top-down design, bottom-up design, stepwise refinement, abstract data types, encapsulation, information hiding, etc., are also used.
    - The design of software is usually expressed in different types of diagrams, each type illustrate an aspect of the software.
- Implementation.
    - Programmers write code for the required computation.
    - Usually programmers develop modules one by one.
- Integration and Testing.
    - Each module is first tested independently so that it matches the specification of the module. This is called **unit testing**.
    - Then the modules are integrated into a software system and the functional and non-functional aspects are tested.
    - Finally, the complete software system is given to users for **user acceptance test (UAT)**.
    - Errors and problems will then be fixed.
- Deployment.
    - The software system is installed for real usage, probably in customer environment, or package for sales online.
    - It may involve more testing and adjustment due to different computer configurations and versioning of operating systems.
- Maintenance.
    - Some problems or errors may surface after the system is running for some time. Patches or bug fixes will be needed.
    - Customers may raise additional requirements and adjustments are to be made to the software system.

Advantages of Waterfall Model

- The stages are very clear and easy to follow.
- Each stage has a definite deliverables and objectives.
- All processes and components are well documented.

Disadvantage of Waterfall Model

- The software system does not appear until in the later stage
    - A lot of risks and uncertainty.
    - Intractable problems in large scale development or innovative products are often hidden in the early stages.

- Difficult to integer large number of modules in one go at the Integration stage.
- Difficult to change design if problems are found at testing stage
- Difficult to accommodate requirement changes.

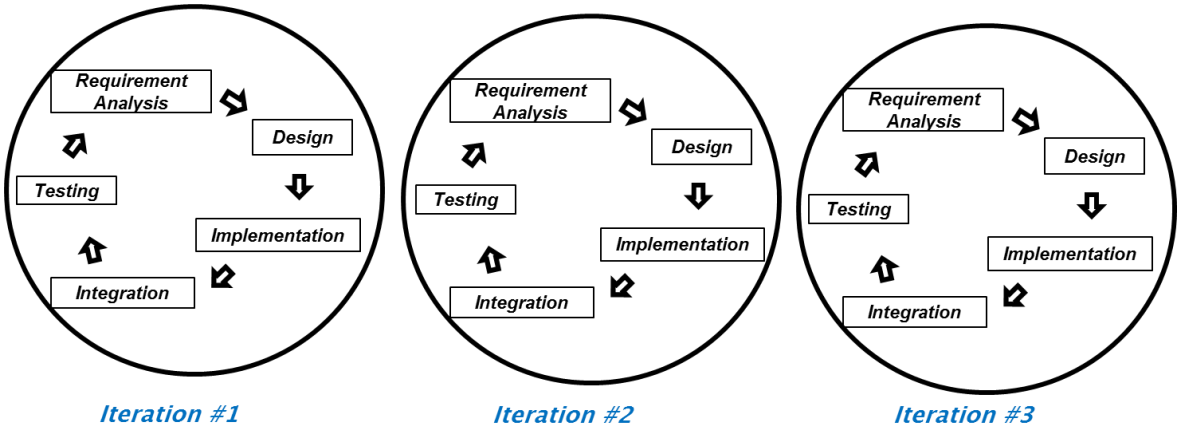
3.4.2 The Agile Model of Software Development

The Agile Model focusses on rapid delivery of software systems through incremental development of one feature and the next.

A coarse illustration of the agile model is to compress the design-implementation-testing-integration cycle in a very short period of time, such as three weeks, but only focus on one feature.

For example, if the functional requirements of a software system include 10 features. The differences between the waterfall model and the agile model are listed below:

- The Waterfall Model
  - Spend several months design and coding modules for the 10 features.
  - Spend a month to integrate the modules into a software system
  - Spend another month to test and deploy the system.
- The Agile Model
  - Spend 3 weeks to work on one feature
  - Spend another 3 week to work on another feature
  - Spend 6 weeks to work on the third feature and so on



Advantages of Agile Model

- Features of software system are rapidly developed, demonstrated and evaluated.
- Promote teamwork.
- Accommodate changes easily.

Disadvantage of Agile Model

- The lack of documentation may cause over-reliant on individuals' knowledge and understanding.
- Needs expertise to drive the practice.

COMPS209F Data Structures, Algorithms and Problem Solving

Recursion 4

Recursion is a specific form of problem solving technique based on divide-and-conquer.

- Modular programming. The divided parts are different in functionality, forms, or purposes.
- Iteration or repetition. The divided parts are very similar (or the same) in functionality, forms, or purposes. It deals with a problem on the ground level with traversal.
- Recursion. The divided parts are very similar (or the same) in functionality, forms, or purposes. It deals with a problem from the top level and works the way down.

Recursion can be implemented as recursive functions programs.

4.1 Principles of Recursion

Recursion is an important problem solving technique based on divide-and-conquer.

- **Divide-and-conquer** handle problems by dividing the problems into sub-problems.
- The sub-problems should be smaller in size or easier to solve.

For recursion to work, the sub-problems should be very similar to the original problem, just smaller in size.

- The sub-problems can be resolved by the same set of logic and computation.
- The aim is to divide the problem until it becomes very simple and a solution is available.

Recursion provides another option of solving a problem. Usually the iterative approach (by repetition and traversal) is another option. However, a small set of problems can only be solvable with recursion (or recursion provides a easier way to solve the problem).

A recursion solution is often implemented with a **recursive function**. The two concepts are different.

- Recursion is a general problem solving technique based on divide-and-conquer.
- Recursive functions are functions that contain a self-calling function call.

4.1.1 Recursive Functions

A recursive function is a function that contains a self-calling function call.

The following function `sumN` is a recursive function. It contains a function call to `sumN()` which is the name of the function itself.

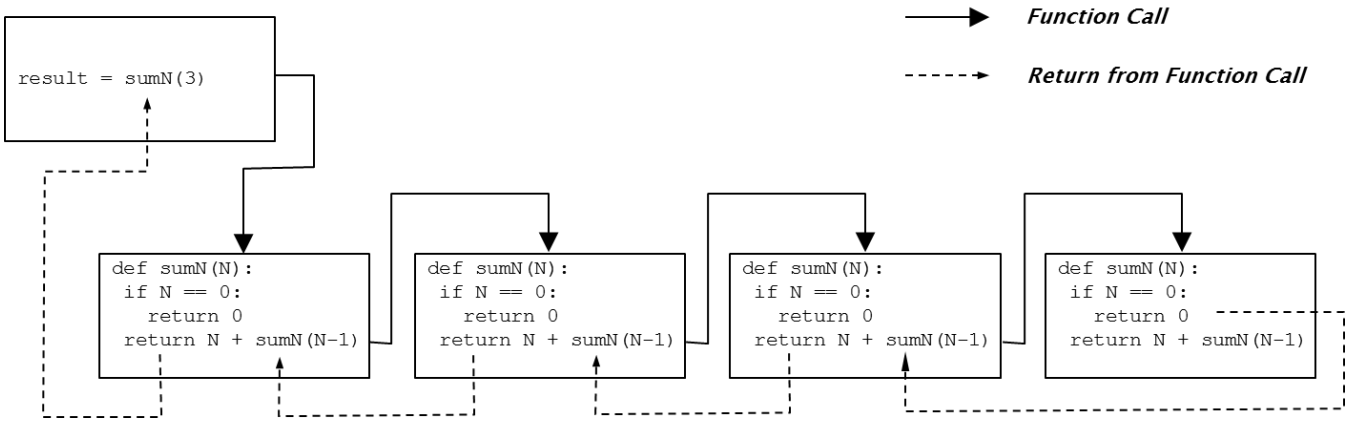
Example: recursive_func_1.py

```
def sumN(N):
    if N == 0:
        return 0
    return N + sumN(N-1)

if __name__ == "__main__":
    result = sumN(100)
    print("The sum of 1 to {} is {}".format(100, result))
```

```
The sum of 1 to 100 is 5050
```

The following shows how a function call to a recursive function will pan out.

- The solid lines indicate the flow of execution during function calls.
- Remember that a function call should normally return. The dashed lines indicate the flow of function call return.
- The point of return can be one of the return statements or end of function body.
- The diagram shows a recursive function calling itself three times.
- At the last time, the `return` statement is executed. The function call returns instead of doing another recursive call of `sumN`.



Good Recursive Functions Should Stop Eventually

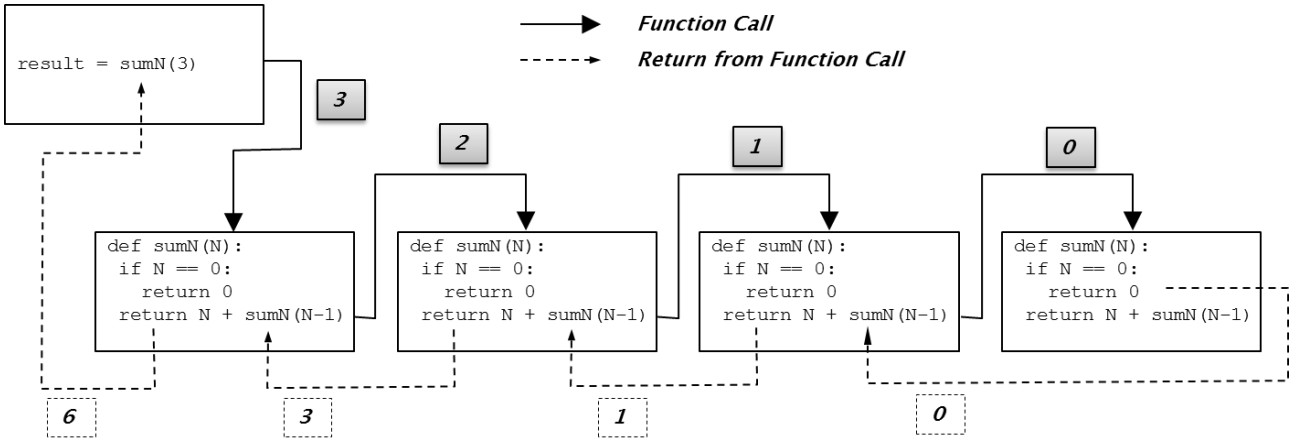
The diagram has illustrated some key principles of recursion,

- A recursive function may be called recursively any number of times.
- A good recursive function should stop calling itself if a condition is met.
    - In this function, the stopping condition is N equals 0.
- A poorly written recursive function may be recursively called indefinite number of times.
    - In this function, it can happen if the initial call is made with a negative parameter N.
    - As the function calls are made, N-1 is passed to the called function.
    - N can never be 0.

Parameter Passing and Return Value in Recursive Functions

The parameters of a recursive function are considered to be the specification of a problem.

- The function sumN should return the sum of 1 + 2 + … + N.
- The N in the parameter specifies the actual problem.
    - For example if N is 10, then the problem is the sum of 1, 2, 3, … and 10.
    - If N is 3, then the problem becomes the sum of 1, 2, and 3.



The above diagram shows the actual values passed in function calls and returned from function calls.

- The function may be called recursively many times, but each time the function solves a *different problem*.
    - At the first time, the parameter N is 3, the function is asked to solve for 1 + 2 + 3.
    - At the second time, the parameter N is 2, the function is asked to solve for 1 + 2.
    - At the third time, the function is asked to solve for 1.
    - At the final time, the function is asked to solve for 0, which means no value in the summation.
- The function should be designed so that it can return a result directly for the simplest version of the problem.
    - For the summation problem, the simplest version is N = 0. The result should be 0.
    - The simplest version is called the **base case**.
    - The base case is always found in a proper recursive function. It contains a condition and a return statement.

```
def sumN(N):
    if N == 0:
        return 0
    return N + sumN(N-1)
```

Simplification of Problems through Recursive Calls

Each recursive call should simplify the problem.

- For the summation problem, the problem is simplified from

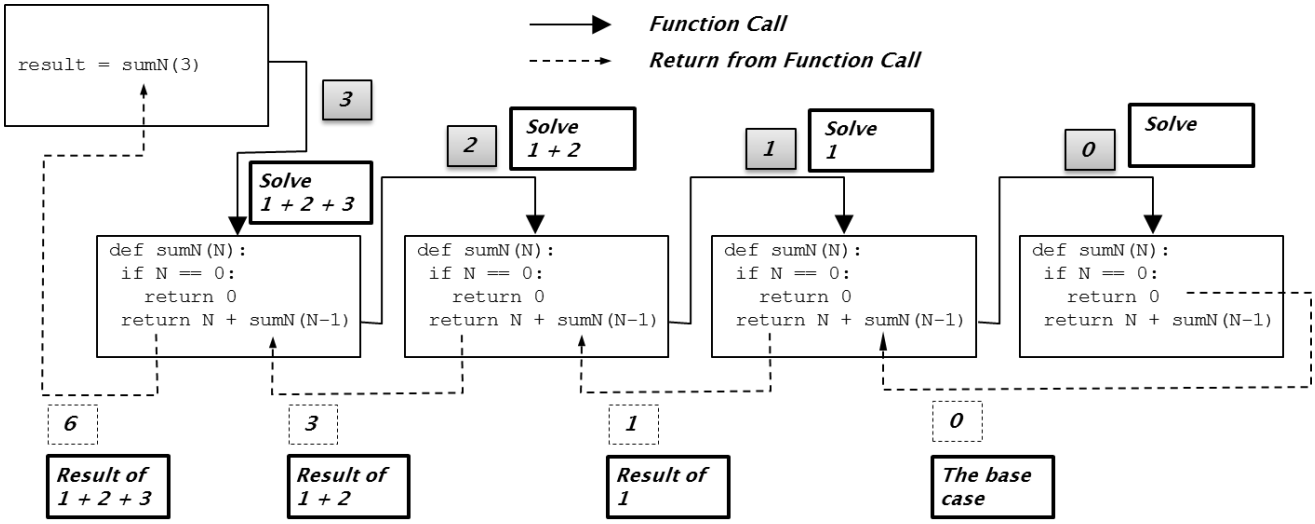
  sum of 1 + 2 + ... + (N – 1) + N to

  sum of 1 + 2 + ... + (N – 1)

- For example: from sumN(3) to sumN(2) is handled through the recursive call.

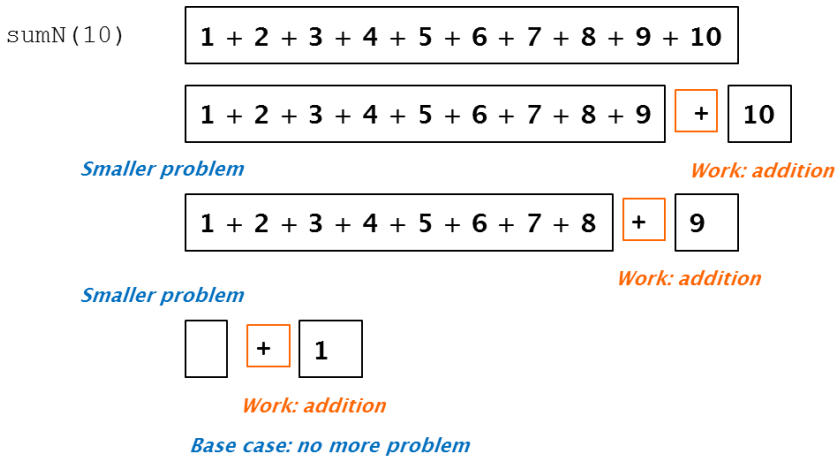
```
def sumN(N):
    if N == 0:
        return 0
    return N + sumN(N-1)
```

The following diagram shows the actual problem solved by each recursive call. Note each recursive call is doing its job of finding the solution of sum of 1 + 2 + …+ N.



Work and Recursive Calls

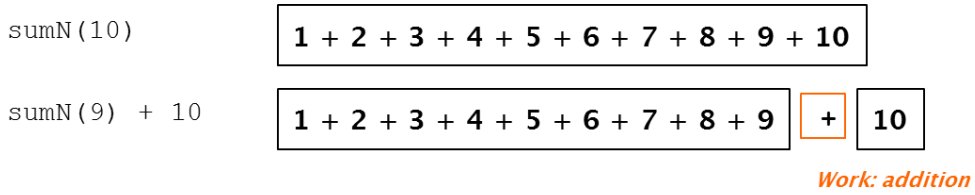
The simplification of problem must come with work. For the summation problem, the work is the summation.



The following statement in the recursive function does the work.

```
    return N + sumN(N-1)
```

The following diagram illustrates how sumN(10) is solved by calling sumN(9) and add 10.



In summary, a recursive function solves problem by three elements:

- Making a recursive call for solving a simplified version of the problem
  - From `sumN(10)` to `sumN(9)`
- Doing work so that the simplification is valid
  - Add 10 to the solution of sumN(9)
- Stopping the recursive call with the base case.
  - The answer for `sumN(0)` is known to be 0.
- The following illustrates how this function can be applied to all the sub-problems.

Solution of sumN(10)

```
    sumN (10)
>>> sumN (9) + 10
>>> sumN (8) + 9 + 10
>>> sumN (7) + 8 + 9 + 10
>>> ...
>>> sumN (0) + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

4.1.2 General Nature of Recursion

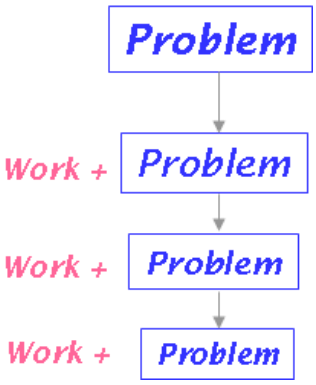
A recursion is a problem-solving method.

Recursion

If the problem is simple and a solution is available, then uses the solution.

If the problem is not simple, divide the problem into one or more sub-problems. The sub-problems should be very similar to the original problem. The sub-problems are then solved by recursion.

Reducing a problem into a smaller problem involves work.



Recursive function is the most suitable means of implementing a recursive solution. Such functions would have a structure similar to the following.

Structure of a Recursive Function

```
Problem Solver {
  If this is a base or simple case and a solution is known,
    Return the solution
  Else {
    Divide the Problem into Sub-Problems.
    Use Problem Solver to get solutions to Sub-Problems.
    Integrate the solutions of Sub-Problems as the solution to the Problem.
  }
}
```

4.1.3 How to Develop a Recursive Solution

Recursion provides a template or a structure for problem solving.

Template of Recursive Functions

Template of a Recursive Function

```
Problem Solver {
  If this is a simple (base) case and a solution is known,
    Return the solution of the base case
  Else {
    Divide the Problem into Sub-Problems.
    Use Problem Solver to get solutions to Sub-Problems.
    Integrate the solutions of Sub-Problems as the solution to the Problem.
  }
}
```

The above structure contains a number of components to be added in. Programmers can then focus on analyzing the problem and finding the suitable components for solving the problem.

1. Specify sub-problems to the original problem that are very similar to the original problem and are progressing towards the base case.
2. Specify the base cases of the problem.
3. Specify the operations (work) to integrate the solutions for the sub-problems into a solution for the original problem.

The smaller problem is the same form as the original problem is significant. The problem and the sub-problems can be solved by the same program/function (no need to rewrite).

Base Case and General Case

Step 2 and 3 above specify the **base case** and the **general case** respectively. The template can be clarified with these two terms.

Refined Structure of a Recursive Function

```
Problem Solver {
Base Case:
  Return the solution of the base case
General Case:
  Divide the Problem into Sub-Problems
  Find the Solutions to the Sub-Problems Recursively
  Integrate the solutions of Sub-Problems as the solution to this Problem
}
```

Recursion shares all characteristics of a divide-and-conquer approach. The characteristics of recursion in the context of divide-and-conquer are summarized in the following table.

Characteristics	***Recursion***
Problem	Any problem that could be divided into sub-problems that are very similar to the original problem (but smaller), and the problem has simple base cases with ready solutions.
Number of Sub-Problems	A problem is always divided into one or more sub-problems that are similar to the original problem.
Size of Sub-Problems	The sub-problem must be smaller than the main problem and getting near to the base case.
Stopping Condition	Stop dividing when the sub-problem becomes a base case.
Method for Integration of Solutions of Sub-Problems	The solution of a problem is evaluated from the solutions of the sub-problems.

Example: Summation of 1 to N

The following table shows the analysis of how to develop a recursive solution for the problem of summation from 1 to N.

Steps	***Remarks***
1. Specify sub-problems to the original problem that are very similar to the original problem	Yes. Summing the list 1 to N is very similar to adding the list 1 to N-1. Summing the list 1 to N-1 is simpler.
2. Specify the base cases of the problem.	Yes. Summing an empty list is 0. Summing a list of 1 to 1 is 1.
3. Specify the operations (the work) to integrate the solutions for the sub-problems into a solution for the original problem.	Yes. Addition of an integer to the sum of a list

4.2 Examples of Recursion

This section discusses several examples of recursion.

4.2.1 Recursion on Sequence of Numbers

Operations on a sequence of numbers are often recursive in nature.

- A shortened number sequence is still a number sequence.
- An operation on a number sequence can be simplified into the same operation on a shortened number sequence.

Example 1: Factorial

The factorial of N is defined as below.

$$fact(N) = 1 \times 2 \times 3 \times \ldots \times N$$

The template is followed to derive the base case and the general case.

Specification

Base Cases:

$fact(0) = 1$

$fact(1) = 1$

General Case:

$$\begin{aligned} fact(N) &= 1 \times 2 \times 3 \times \ldots \times N \\ &= 1 \times 2 \times 3 \times \ldots \times (N-1) \times N \\ &= fact(N-1) \times N \end{aligned}$$

The following shows an implementation.

Example: factorial.py

```
def fact(N):
    if N == 0:
        return 1
    elif N == 1:
        return 1
    return fact(N-1) * N

if __name__ == "__main__":
    result = fact(10)
    print(result)  # factorial of 10
    result = fact(50)
    print(result)  # factorial of 50
```

```
3628800
30414093201713378043612608166064768844377641568960512000000000000
```

Example 2: Fibonacci number

The original definition of Fibonacci numbers is already in a recursive form.

Specification

Base Case:

fibonacci (0) = 0

fibonacci (1) = 1

General Case:

fibonacci (N) = fibonacci (N-1) + fibonacci (N-2)

The following shows an implementation.

Example: fibonacci.py

```
def fibonacci(N):
    if N == 0:
        return 0
    elif N == 1:
        return 1
    return fibonacci(N-1) + fibonacci(N-2)
```

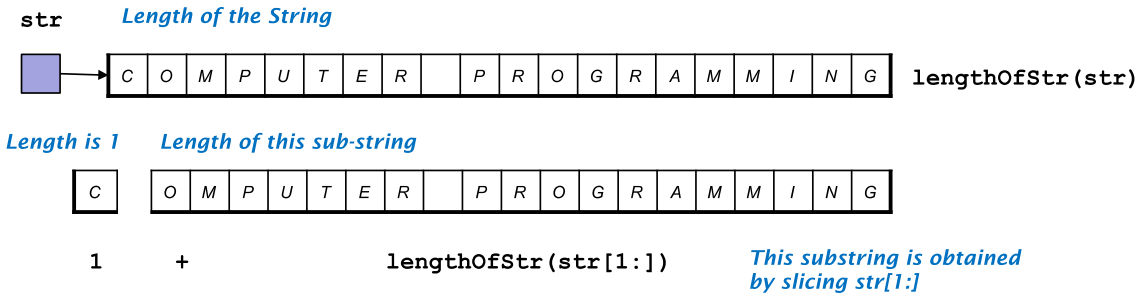
```
if __name__ == "__main__":
    result = fibonacci(10)
    print(result)  # factorial of 10
    result = fibonacci(20)
    print(result)  # factorial of 20
```

4.2.2 Recursion on a Sequence

Sequence data types include strings, lists, and tuples. Operations on a sequence are also recursive in nature. A part of a sequence is also a sequence. Usually a part of a sequence is called a sub-sequence.

Example 1: Length of Strings (Slicing Version)

A string is a sequence. Given a string, finding the length of a string can be formulated as a recursive function.



The following shows an implementation.

Example: string_len.py

```
def lenOfString(str):
    if not str:
        return 0
    return 1 + lenOfString(str[1:])
```

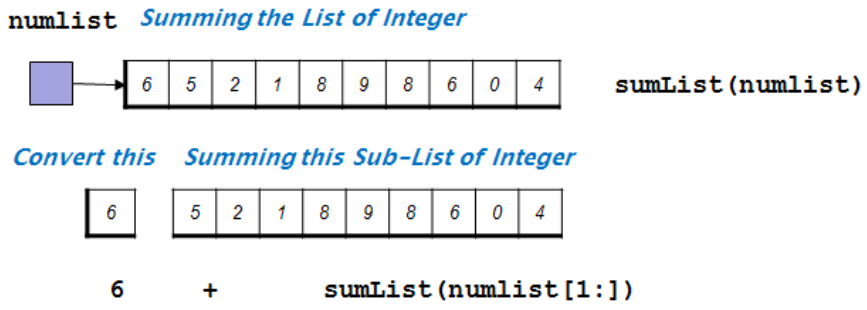
```
if __name__ == "__main__":
    s1 = "COMPUTER PROGRAMMING"
    result = lenOfString(s1)
    print(result)
```

Notes:

- The slicing operation creates a new string at every recursive call.
- The slicing operation consumes additional time and memory.

Example 2: Sum a List of Integers (Slicing Version)

Summing a list of integers means adding each integer in a list to a sum one by one.



The following shows an implementation.

Example: integer_list.py

```
def sumList(numlist):
    if not numlist:
        return 0
    return numlist[0] + sumList(numlist[1:])
```

```
if __name__ == "__main__":
    numlist = [6, 5, 2, 1, 8, 9, 8, 6, 0, 4]
    result = sumList(numlist)
    print(result)
```

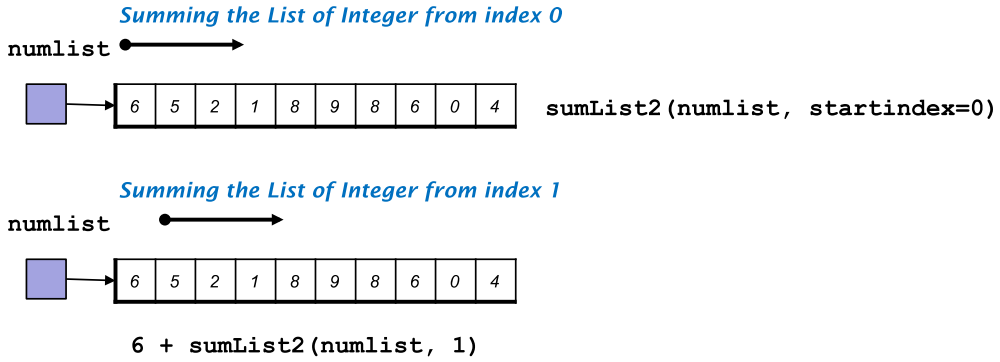
Notes:

- The slicing operation creates a new list at every recursive call.

Example 3: Sum a List of Integers (Passing Index Version)

To avoid creating a new list at each recursive call, the recursive function may be re-designed as follows:

- Always accept the same list at every recursive level.
- Handle different part of the list with an index parameter.



The following shows an implementation.

Example: integer_list.py

```
def sumList2(numlist, startindex=0):
    if startindex == len(numlist):
        return 0
    return numlist[startindex] + sumList2(numlist, startindex+1)
```

```
if __name__ == "__main__":
    numlist = [6, 5, 2, 1, 8, 9, 8, 6, 0, 4]
    result = sumList2(numlist)
    print(result)
```

Notes:

- The recursive function `sumList2` has a default named parameter `startindex`.
- The parameter `startindex` specifies which part the list is to be summed.
    - If `startindex` is 0, then all integers from index 0 to the last index are to be summed.
    - The parameter is added one by one so that the part of the list to be summed is getting smaller – simplification of the problem.
- The stopping condition is the `startindex` reaches the end of the list.
    - There is nothing left in the list to be summed.

4.3 Merits and Drawbacks of Recursion

Many problems can be solved with both iteration and recursion.

- Iteration means solving a problem with procedural repetition structure such as a while loop.
- Recursion means solving a problem with recursive calls.

The following compares an iterative solution and a recursive solution of summation from 1 to N.

Iterative

```
def sumN(N):
    sum = 0
    for num in range(1, N+1):
        sum += num
    return sum
```

Recursive

```
def sumN(N):
    if N == 0:
        return 0
    return N + sumN(N-1)
```

Iterative solutions are procedure based:

- They are based on a repetition structure (traversal).
- Repetition continues until the loop-continuation condition fails.
- Make progress by repetition.
- Loop-continuation condition will become false eventually.

Recursive solutions are specification based:

- They are based on self-calling that continues until a base case is reached.
- Make progress by solving simpler versions of the original problem.
- The recursion continues until the base case is reached.

Recursion is not a simple programming topic. However, once inexperienced programmers can solve problems with the recursive approach, they have a tendency to use recursion most of the time. They should be aware of the merits and drawbacks of recursion.

Merits of Recursion:

- Simple to implement if base cases and general cases are identified
- The program structure looks clearer
- Often easier to implement than using iterative approach

Drawbacks of Recursion:

- Slower to execute. Due to function calls using additional time.
- Consume more stack memory. The amount of stack memory may run out.

4.4 Stack Memory Limitation and Tail Recursion

Each recursive call is a function call that would create local variable set in the stack memory.

Consider the following example.

Example: recursive_stack.py

```
import sys

def sumN(N):
    if N == 0:
        return 0
    return N + sumN(N-1)

if __name__ == "__main__":

    print("Maximum recursive limit =", sys.getrecursionlimit())

    N = 80
    result = sumN(N)
    print("sumN({}) = {}".format(N, result))

    N = 1000
    result = sumN(N)
    print("sumN({}) = {}".format(N, result))
```

```
Maximum recursive limit = 1000
sumN(80) = 3240
Traceback (most recent call last):
...
RecursionError: maximum recursion depth exceeded in comparison
```

Notes:

- Python has a guard on the maximum level of recursive call.
- When the maximum is reached, the execution will stop and an error of `RecursiveError` is raised.
- The maximum recursion level can be obtained with the following.

  ```
  sys.getrecursionlimit()
  ```

- The maximum level can be adjusted with the following.

  ```
  sys.setrecursionlimit(1200)
  ```

Tail Recursion

Tail recursion means that a recursive function is based on a **tail call**. A tail call is a self-function call that happens to be the last instruction or action of a recursive function.

Consider the following recursive functions for summation from 1 to N.

Regular Recursive Function

```
def sumN(N):
    if N == 0:
        return 0
    return N + sumN(N-1)
```

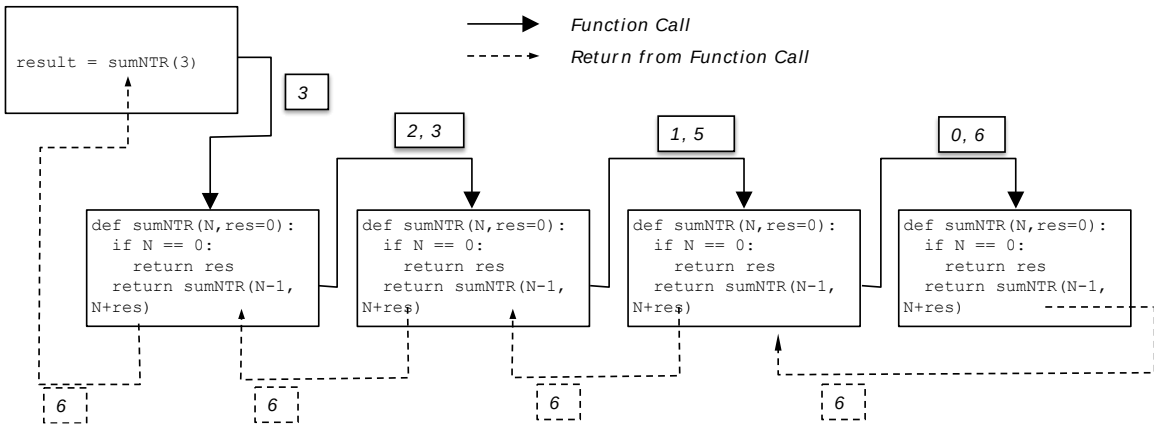
Tail Recursion

```
def sumNTR(N, res=0):
    if N == 0:
        return res
    return sumNTR(N-1, N + res)
```

Notes:

- For the regular recursive function version, the last action is the summation.
    - The summation happens after the self-function call.
- For the tail recursion version, the last action is the self-function call.
    - The recursive function has an additional named parameter `res`. The parameter is used to pass the result of summation with each recursive call.
    - The summation happens before the self-function call.

The following figure illustrates how the tail recursive function `sumNTR` passes the parameters and returns the result.



Notes:

- The partial result of summation is passed through the parameter `res`.
- At the last recursive call, the parameter `res` contains 6. This is the result of the summation of 1 to 3.
- This final result of 6 is then passed back as the return value.

Tail recursion is significant because some programming languages, especially the functional programming languages, have a feature called **tail recursion elimination**. This feature is an optimization done by the compiler that can turn a recursion into an equivalent iterative loop.

- The speed can become comparable to a loop
- No more limitation caused by the stack memory.

Unfortunately, Python does not support tail call optimization (TCO) or tail call elimination. However, the C programming language does support this optimization. Java 8's functional programming interface also supports lambda function and TCO.

> ❖ ***Reasons of No Tail Recursion Optimization in Python***
>
> Tail recursion optimization is not supported in Python. This design has been subject to a lot of debate for many years. For those who are interested in the rationale, read the following articles.
>
> - http://neopythonic.blogspot.com.au/2009/04/tail-recursion-elimination.html
> - http://neopythonic.blogspot.com.au/2009/04/final-words-on-tail-calls.html

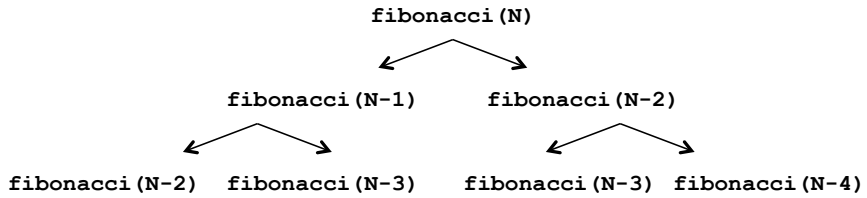
4.5 (Challenging) Speed Up Recursion with Dynamic Programming

Consider the recursive implementation for the Fibonacci function again.

Example: fibonacci.py

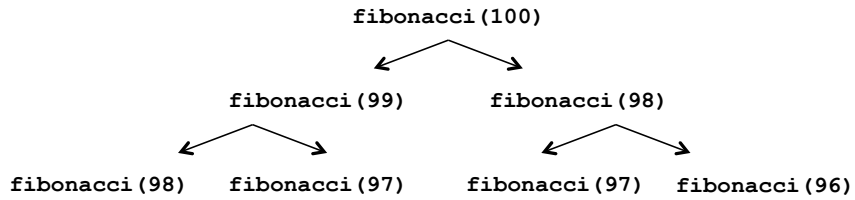
```
def fibonacci(N):
    if N == 0:
        return 0
    elif N == 1:
        return 1
    return fibonacci(N-1) + fibonacci(N-2)
```

The recursive implementation of Fibonacci number is especially time consuming, because two recursive calls are induced by each recursive call.



If the above recursive call tree is examined closely, some recursive calls are carried out more than once. For example, if N is 100, then the following call tree shows that `fibonacci(98)` is called twice.

- A call to `fibonacci(98)` is already time consuming.
- The first time may be unavoidable.
- The second time is definitely wasting time.



Some time can be saved if the following is done:

- When `fibonacci(98)` is called the first time, the result is saved in a list.
- When `fibonacci(98)` is called the second time, the result is looked up from the list.

The following implementation of Fibonacci number generator makes use of a list to store the results of different Fibonacci numbers for lookup.

Example: fibonacci_dp.py

```
def fibonacciDP(N, result=[]):
    # extend the result list and fill with 0
    result.extend([0] * (N - len(result) + 1))

    # base case
    if N == 0:
        return 0
    elif N == 1:
        return 1

    # lookup the list for results
    if result[N-1] == 0:
       result[N-1] = fibonacciDP(N-1, result) # if fib(N-1) not found
    if result[N-2] == 0:
       result[N-2] = fibonacciDP(N-2, result) # if fib(N-2) not found

    # calculate fib(N) and store the result in the list
    result[N] = result[N-1] + result[N-2]
    return result[N]
```

Notes:

- The named parameter result is used to store the partial results (i.e. Fibonacci numbers less than N) for later lookup. A list is used. The index of the list represents the rank of the Fibonacci number.
- Before making a recursive call, a lookup on the result list is done first.
    - The recursive call is made only if there is no result found (if the relevant element is 0).
    - After the recursive call, the Fibonacci number is stored into the result list.
- The result list remains the same object – it is only created once when the function `fibonacciDP` is first called.
    - The Fibonacci numbers stored in the result list are available in future calls to `fibonacciDP`.

Dynamic Programming

The above example is a simplified illustration of a general optimization technique called Dynamic Programming (DP). Dynamic programming means *nesting smaller decision problems inside larger decisions*, according to Wikipedia.

Dynamic programming is therefore a problem solving approach based on the following concepts:

- Break a large problem into smaller problems recursively.
- Avoid solving the same problem again.
- Prefer lookup solutions instead of solving problems.
- Save the solution of problems for later lookup.

COMPS209F Data Structures, Algorithms and Problem Solving

Searching Algorithms 5

Searching for data is a very common computer operation. Searching algorithms are methods for locating and retrieving desired data. Here are the common types of data searching:

Tasks	Examples	Remarks
Search if a certain data exists	The name "Andrew" exists in a list?	The outcome is True or False
Search for a data of some particular properties	What are the largest and the smallest numbers?	
Search for the location that contains a particular data	What is the location of the largest number in the list?	The outcome is the location or the index within a list
Searching for the data associated with a key	What is the phone number of a student with OUID 12345678 (i.e. the key)?	

5.1 Principles of Data Searching

Data searching is a process of two elements:

- Examine each data one by one.
- Find and report the answer.

Python and other programming languages provide functions for many data searching tasks. The following example shows data searching tasks on population information of US cities.

Assume that the information of major US cities, which is stored in a text file, contains the name of the city, the state where the city is located and the population. Each line contains information of one city, with the three items separated by a comma. This is commonly called comma separated list. The first line contains header information.

Text File: us-cities.csv

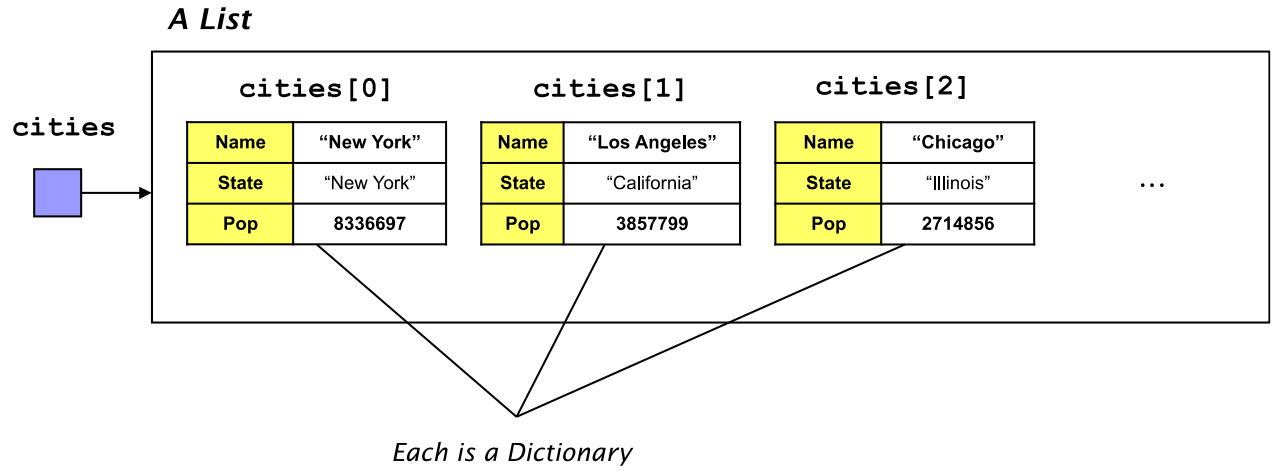
```
City,State,Population
New York,New York,8336697
Los Angeles,California,3857799
Chicago,Illinois,2714856
Houston,Texas,2160821
Philadelphia,Pennsylvania,1547607
Phoenix,Arizona,1488750
San Antonio,Texas,1382951
San Diego,California,1338348
Dallas,Texas,1241162
```

5.1.1 Building a Data Structure for Searching

The first principle: data searching should be carried out in an in-memory data structure.

- Normally data is stored permanently in a file or somewhere in the Internet.
- Reading data from a file or network is a comparatively slow operation.
    - Reading once is inevitable. Avoid reading again and again from the file.
- Building a data structure for efficient searching.
    - Use a combination of lists, dictionaries, arrays, etc., that store data in memory.

Using a list of dictionaries is a common way to handle data records with a key. The key for the US city dataset is the city name, and the values are the state and the population of the city. The following shows how information of US cities is organized in a list of dictionaries. Information of each city is stored in one dictionary, and the dictionaries are stored in a list.



The following shows a Python program for reading the data from a file and building a list of dictionaries.

Example: us-cities.py

```
cities = list()
isFirstLine = True
try:
    with open('us-cities.csv','r') as infile:
        while True:
            line = infile.readline()
            if line:
                if isFirstLine:    # skip the first line
                    isFirstLine = False
                    continue
                row = line.rstrip().split(',')  # rstrip and split line into a list
                city = dict()  # create a dictionary for one city
                city['Name'] = row[0]
                city['State'] = row[1]
                city['Pop'] = int(row[2])
                cities.append(city)
            else:
                break
except IOError:
    print("Error in opening file")
    sys.exit()
```

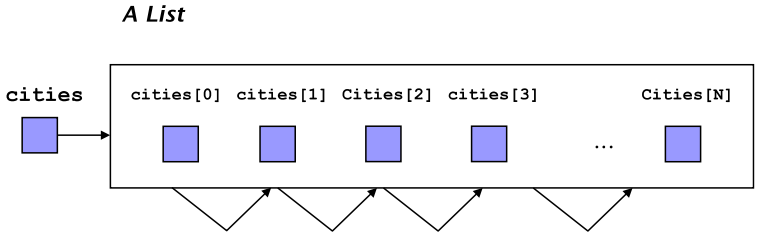
Notes:

- The variable `cities` is the list. It contains many dictionaries, each contains information of one city.
- Each of the dictionaries contains three fields with names `Name`, `State`, and `Pop`, which contains the name, the state, and the population of the cities. Note the population is an integer.
- The variable `isFirstLine` is used to record if this is the first line from the file.

5.1.2 Searching Data in a List of Dictionaries

The second principle: searching for data in a data structure is done by traversal.

The city information is stored in a list one city by another city. Searching is also carried out with examining each city in the list one by one.



- Traversal through the list one element by another element.
- At each element, the dictionary is referred and its content is examined.

Using a For Loop

The following example shows how to use a for loop to find out the city with the longest name in the dataset.

Example: us-cities-search-2.py

```
cities = list()
# OMITTED: data loading from file

# traversal through the list
longestCity = None
for city in cities:
    if longestCity is None or len(city['Name']) > len(longestCity['Name']):
        longestCity = city

print("The city with the longest name is", longestCity['Name'])
```

```
The city with the longest name is San Buenaventura (Ventura)
```

Using a While Loop

The following example shows how to use a counter-controlled while loop to find out the largest city in the state of Ohio.

Example: us-cities-search-3.py

```
cities = list()
# OMITTED: data loading from file

# traversal through the list
cityCount = len(cities)
largestInOhio = None
i = 0

while i < cityCount:
    if cities[i]['State'] == "Ohio":
        if largestInOhio is None or cities[i]['Pop'] > largestInOhio['Pop']:
            largestInOhio = cities[i]
    i += 1

print("The largest city in Ohio is {} with population {}".format(largestInOhio['Name'],
largestInOhio['Pop']))
```

```
The largest city in Ohio is Columbus with population 809798
```

Notes:

- The counter `i` goes through all index of the list.
- Each element in the list is referred by `cities[i]`. Remember that each element is a dictionary.
- The State field of the dictionary is referred by `cities[i]['State']`.

Using Python Function max() and min()

The Python built-in function max() offers a convenient way to find the maximum element in a list. Similarly the function min() is also available. However, the elements in the list must be *comparable*.

- Comparable means there is a natural order of larger and smaller.
- Numbers like integers and floating points are clearly comparable.
- Strings are also comparable based on the ASCII (or UTF) encoding.
- Dictionaries and other objects are however not comparable.

The following shows examples of searching for maximum and minimum of lists of numbers and strings.

Example: search-int-list.py

```
numlist = [10, 5, 3, 15, 32, 20, 9, 6]
maxnum = max(numlist)
minnum = min(numlist)
print(maxnum, minnum)  # prints 32 3

strlist = ['Besty', 'Anders', 'Doris', 'Chris']
maxstr = max(strlist)
minstr = min(strlist)
print(maxstr, minstr)  # prints 'Doris' 'Anders'
```

Using Python Function max() and min() with the argument key

For lists of non-comparable elements, the functions max() and min() offer a keyword argument for specifying a function that provides the comparable.

```
max(list, key=<field>)
```

The following shows how to define a function for finding the city with the maximum Population.

Example: us-cities-search-4.py

```
def getPop(city):
    return city['Pop']
```

```
...
# Using the key argument for specifying the field of dictionary for comparison
largestCity = max(cities, key=getPop)
print("The largest city in US is {}".format(largestCity['Name']))
```

Notes:

- The key argument specifies a function called `getPop`.
- The function `getPop` must accept an element of the list and return a comparable based on the element.
- The function `getPop` is called by the function max() repeatedly through traversing the elements in the list.
- Remember that each element in the list `cities` is a dictionary containing information of a city. This element is passed through the parameter **`city`** in the function **`getPop`**.
- The function returns the `Pop` field of the dictionary.

More sophisticated logic can be added to the key function. The following function only returns the population of cities of which the state is 'Ohio'. It returns -1 for cities of other states. Because only cities of the state 'Ohio' return a positive number, the max() function only considers these cities.

Example: us-cities-search-4.py

```
def getPopOfOhio(city):
    if city['State'] == 'Ohio':
        return city['Pop']
    else:
        return -1
```

```
# Finding the largest city in the state Ohio
largestInOhio = max(cities, key=getPopOfOhio)
print("The largest city in Ohio is {} ".format(largestInOhio['Name']))
```

If the above function is used with max(), the largest city in the state Ohio is found.

Using Python Function max() and min() with lambda functions

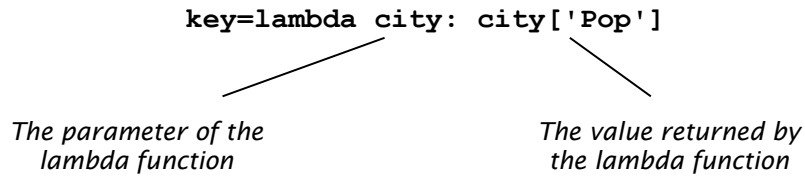
The lambda function definition provided by Python allows inline definition of the key function.

The following example shows the definition of a lambda function for returning the 'Pop' field of the city dictionary.

Example: us-cities-search-4.py

```
# Using the lambda fucntion for more sophisticated queries

largestCity = max(cities, key=lambda city: city['Pop'])

print("The largest city in US is {}".format(largestCity['Name']))
```

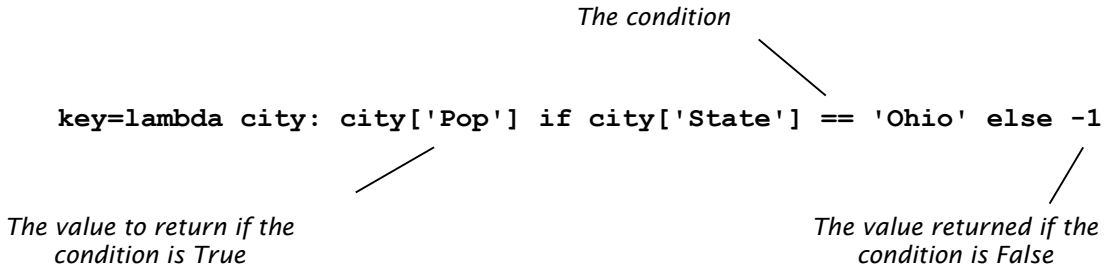
The following illustrates the structure of the lambda function definition.



An optional if-else structure may be added to the lambda function definition. The following query finds the largest city in the state 'Ohio'.

Example: us-cities-search-4.py

```
# Using the lambda fucntion for more sophisticated queries

largestInOhio = max(cities, key=lambda city: city['Pop'] if city['State'] == 'Ohio'
else -1)
print("The largest city in Ohio is {} with population {}".format(largestInOhio['Name'],
largestInOhio['Pop']))
```



5.1.3 Importance of Data Organization for Searching

Finding maximum or minimum in a list of elements normally requires the traversal of the whole list.

However, searching is actually not required if the data in the list is sorted.

For a sorted list in ascending order (i.e. from small to large), the first element is the minimum and the last element is the maximum. No searching is needed.

The following shows an example of how sorting a number list makes finding maximum and minimum simple.

Example: search-sorted-list.py

```
numlist = [10, 5, 3, 15, 32, 20, 9, 6]

numlist.sort()

print(numlist) # print [3, 5, 6, 9, 10, 15, 20, 32]

print("The minimum number is", numlist[0])
print("The maximum number is", numlist[-1])
```

```
[3, 5, 6, 9, 10, 15, 20, 32]
The minimum number is 3
The maximum number is 32
```

Notes:

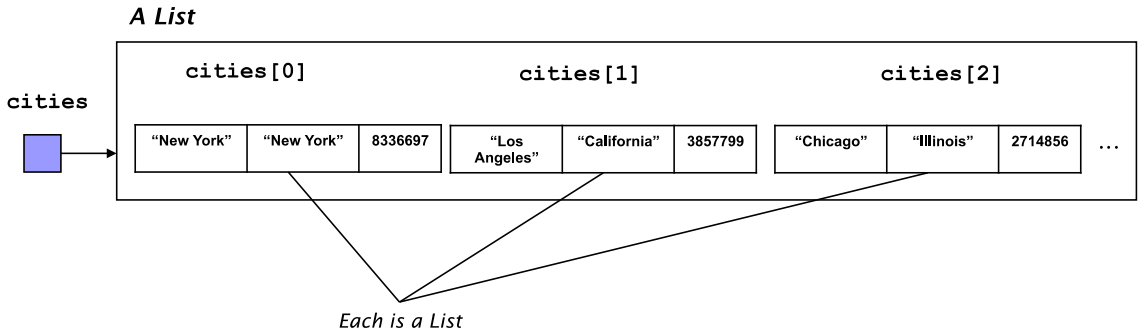
- A sorted list is efficient for finding minimum and maximum.
- Sorting is however a time consuming operation.
- Therefore, a sorted list is useful to keep it sorted. Data should be added to a sorted list in a way that the list will remain sorted.

The third principle: certain data organization, such as sorted lists, can facilitate searching.

5.1.4 An Alternative: Using a List of List

The list of dictionaries is clearly not the only method to organize data. There are many other ways.

One alternative is to have the US city population dataset stored in memory using a list of list (instead of a list of dictionaries).



The following shows an example how to use a list of list for data searching.

Example: us-cities-search-5.py

```
cities = list()
isFirstLine = True
try:
    with open('us-cities.csv','r') as infile:
        while True:
            line = infile.readline()
            if line:
                if isFirstLine:    # skip the first line
                    isFirstLine = False
                    continue
                row = line.rstrip().split(',')  # apply rstrip and split into a list
                # row[0] is name, row[1] is state, and row[2] is population
                row[2] = int(row[2]) # convert population into integer
                cities.append(row)
            else:
                break
except IOError:
    print("Error in opening file")
    sys.exit()

# print(cities)

# Using the lambda function for more sophisticated  queries
largestCity = max(cities, key=lambda city: city[2])
print("The largest city in US is {}".format(largestCity[0]))

largestInOhio = max(cities, key=lambda city: city[2] if city[1] == 'Ohio' else -1)
print("The largest city in Ohio is {} with population {}".format(largestInOhio[0],
largestInOhio[2]))
```

```
The largest city in US is New York
The largest city in Ohio is Columbus with population 809798
```

Notes:

- The fields of a city's name, state, and population are now stored in a list. Every one of the lists is added to the list of cities referred by the variable `cities`.
- The population field is `row[2]`, and it is converted to integer so that it can be comparable.

This alternative is functionally equivalent to the method of list of dictionaries. The drawback is poorer readability. For example, `city['Pop']` is easier to understand than `city[2]`. Dictionaries have named keys and therefore their meanings are more visible.

5.1.5 Another Alternative: Not Storing Data In-Memory

A data structure is not necessary for data searching. Another alternative is to read data from file and to search data at the same time.

Example: us-cities-search-fromfile.py

```
largestCity = largestInOhio = None
isFirstLine = True
try:
    with open('us-cities.csv','r') as infile:
        while True:
            line = infile.readline()
            if line:
                if isFirstLine:    # skip the first line
                    isFirstLine = False
                    continue
                row = line.rstrip().split(',')  # apply rstrip and then split into a list
                row[2] = int(row[2]) # convert population into integer

                if largestCity is None or row[2] > largestCity[2]:
                    largestCity = row
                if row[1] == 'Ohio' and (largestInOhio is None or row[2] > largestInOhio[2]):
                    largestInOhio = row
            else:
                break
except IOError:
    print("Error in opening file")
    sys.exit()

print("The largest city in US is {}".format(largestCity[0]))
print("The largest city in Ohio is {} with population {}".format(largestInOhio[0],
largestInOhio[2]))
```

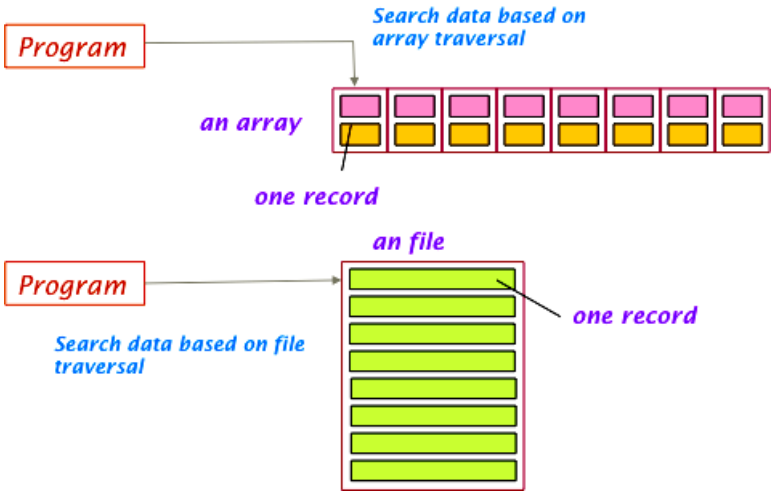
```
The largest city in US is New York
The largest city in Ohio is Columbus with population 809798
```

The bold lines in the program contain logic to search for the largest city in US and the largest city in the state of Ohio.

Notes:

- The variables `largestCity` and `largestInOhio` are updated while reading in the city data one record by another record.
- No data is stored in-memory.

This method is not suitable if the program needs the dataset for other operations. For example, if the program allows the user to repeatedly query the dataset, then the dataset must be stored in-memory to support this operation.



5.2 Efficiency of Data Searching

Data searching can be a time consuming process if the data amount is large.

- Searching involves examining each data one by one.
- If there are a million elements in a list, then there are a million elements to examine.
- The time of searching is reduced if not all data needs to be examined.

Generally, the efficiency of any algorithm, including those for data searching, can affect the performance of any program or system. The time taken to execute an operation should be a big concern of programmers. There are two major approaches of efficiency analysis:

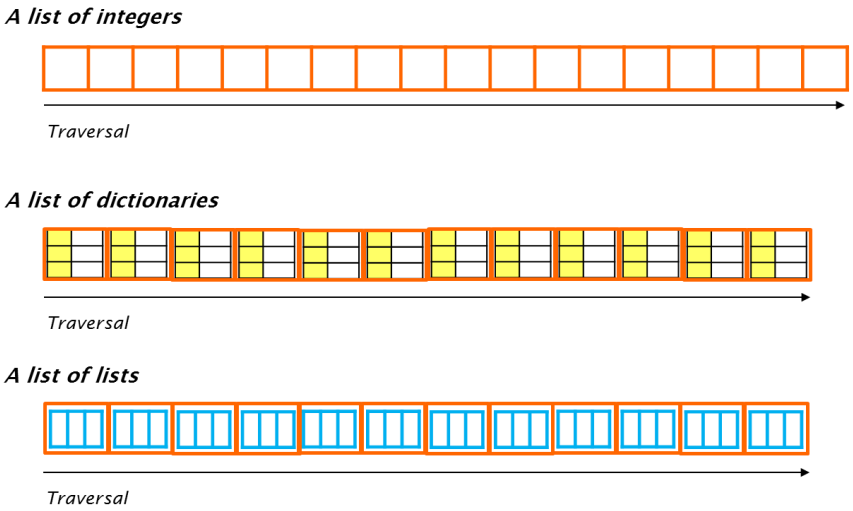
- Analytical approach: estimating the efficiency of a method or an algorithm by analyzing the code and the operation patterns. The time taken is estimated with counting how many repetitions to occur or how many comparisons will take place.
- Empirical approach: estimating the performance of a method or an algorithm by carrying experiments to measure the performance. Basically, the actual time taken in data searching is recorded and analysis.

5.2.1 Sequential Search

Sequential search or **linear search** is a searching operation that traverses a data structure and examines each data one by one. The examination of data is performed in a sequential manner, and so the algorithm named.

Sequential search is also known as linear search because the search is performed on data structure in a line or linearly. It is one of the **searching algorithms**.

As discussed in the previous section, traversal and matching are the two main actions involved in a searching algorithm. Traversal in this context means accessing and examining data one by one. The data can be in a file (line by line) or in a data structure in memory (element by element). For each data in the traversal, the data is examined and checked if it matches the searching objective.



Efficiency of data searching is not a big deal if the amount of data is small.

Consider a very slow computer that can search one record of HK citizen with the HKID for 0.000001 seconds. It sounds not too slow for small amount of data, but a big deal for large amount of data.

- If there were only 10 HK citizens, then a full traversal will take 0.000001 x 10 = 0.00001 seconds.
- If there are 8 million HK citizens, then it will take 0.000001 x 8000000 = 8 seconds

---

5.2.2 Efficiency of Algorithms: Analytical Approach

The analytical approach estimates the efficiency of an algorithm through analysing the code. There are a number of important concepts.

Large Amount of Data

As described in the HKID example above, efficiency is an important consideration mostly when there are large amount of data. Any analysis should focus on the cases when the data amount is large.

Scalability

The **scalability** of an algorithm is how the algorithm can handle increasing amount of data.

- An algorithm is said to scale well means that the algorithm continues to perform reasonably with increased data amount.
- An algorithm does not scale if the algorithm does not perform reasonably when given a large amount of data.

The scalability is often analysed based on consideration of two cases: having N elements and having many times of N elements (such as 2 times N, 4 times N, etc.).

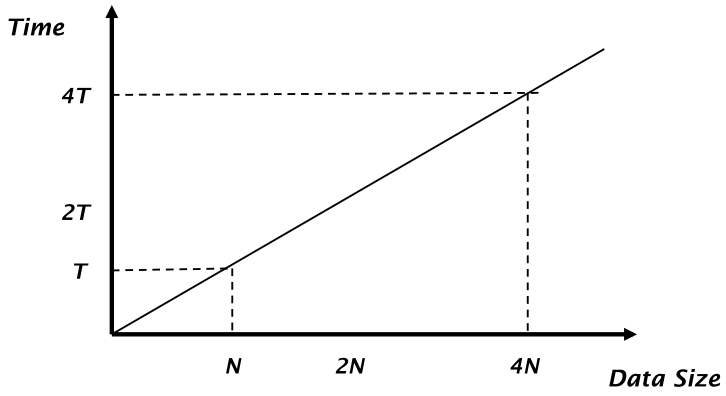
- The N element case offers the baseline performance. The baseline is the standard for comparison.
- The 2N element case (i.e. 2 times N) means a larger amount of data.
- The 4N element case and beyond means even larger amount of data.
- The performance of the N elements is then compared to the 2N, 3N, 4N elements and so on.

Linear Scalability

If a carpenter needs 1 day to build a table, then logically he or she will take 100 days to build 100 tables. This efficiency or performance of the carpenter is acceptable in our world.

In the same sense of acceptable observation, if an algorithm takes time T to handle N amount of data, then it should take 2T to handle 2N amount of data.

- The algorithm is considered to **scale linearly** with the amount of data.
- Linear scalability means that the time taken is proportional to the amount of data.



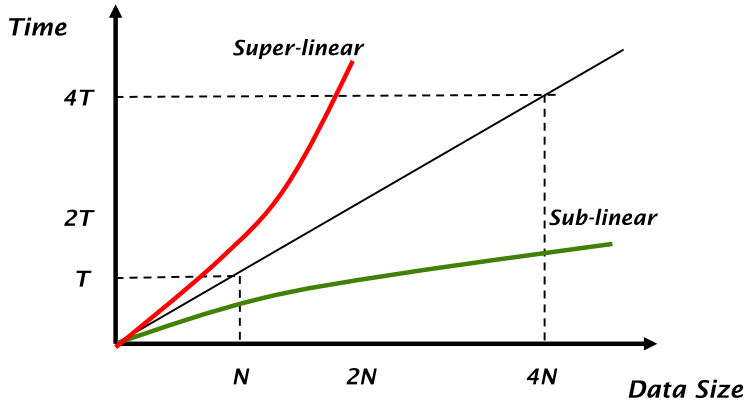
Sub-linear and Super-linear Scalability

The following two cases illustrate sub-linear and super-linear scalability.

- Sub-linear scalability: a carpenter takes 1 day to build 1 table, and 1 day also to build 10 tables.
- Super-linear scalability: a carpenter takes 1 day to build 1 table, but 10 days to build 2 tables.

These two cases do not normally happen with carpenters, but they can happen with computer algorithms.

- Sub-linear scalability algorithms: scale well such that the increment in time taken is slower than the increment in data size. These algorithms are desirable.
- Super-linear scalability algorithms: do not scale well that the increment in time taken is faster than the increment in data size. These algorithms are not so desirable.



If an algorithm exhibits performance characteristics similar to the sub-linear (green) line, then the algorithm is desirable. Although the data size increases from N to 4N (i.e. 4 times), the time taken only increases from T/2 to T (approximately 2 times).

Worst-Case, Best-Case, and Average-Case

Sometimes the time taken of an algorithm is affected by the particular dataset.

For example, consider searching for HKID A123456(7) from a list of dictionaries containing information of HK citizens. The dictionary containing the given HKID can occur anywhere in the list.

- If the given HKID locates in the very first element in the list, then the data searching will be the fastest. The number of checking required is only 1. This is a very lucky case. This is called the **best case**.
- If the given HKID locates in the very last element in the list, then the data searching will involve checking all elements. The number of checking required is same as the data size. This is a very unlucky case. This is called the **worst case**.

The best case and the worst case represent the upper bound and the lower bound performance. If we know the best-case performance and the worst-case performance, the actual performance must be somewhere in between.

The **average case** performance indicates usually how an algorithm would perform. This represents the average performance after running the algorithm on many datasets.

Importance of the Worst-Case Performance

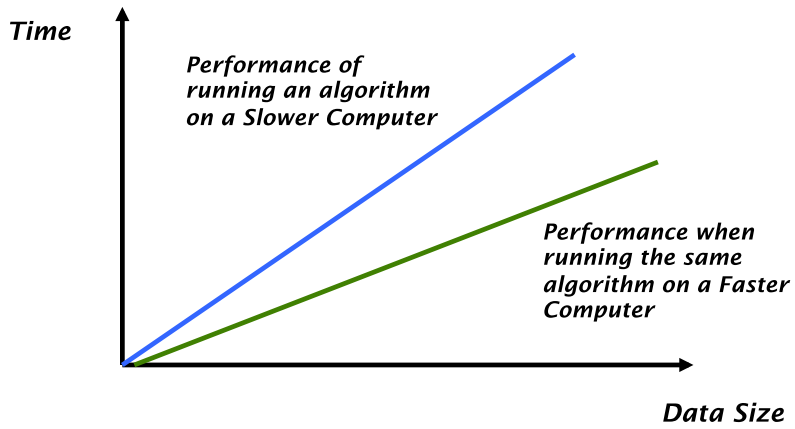
It is interesting that software developers, programmers, and engineers are always interested in the worst-case scenarios.

- Worst case represents the lower-bound performance.
- If the worst case of an algorithm is deemed to be acceptable, then the confidence level that the algorithm producing acceptable performance is very high.

Independence of Computer Speed

The speed or the power of a computer will affect the time taken of an algorithm handling a dataset. A more powerful computer should run faster.

- The efficiency of an algorithm should be considered independent of the speed of computers.
- The target of analysis is the algorithm alone.



The above figure illustrates the performance characteristic of the same algorithm running on two different computers. The faster computer will give an absolute shorter time taken. However, the scalability of the algorithm is always the same, and in this case scaling linearly.

5.2.3 Efficiency Evaluation of Sequential Search: Analytical Approach

The best case and the worst-case performance of sequential search are very much different.

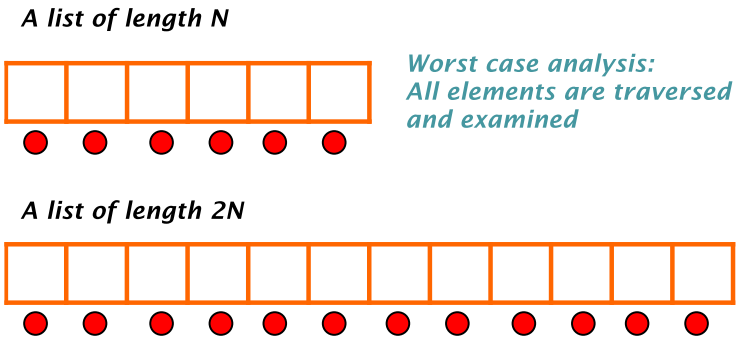
Consider that the data size is N, the number of checking required to complete the searching operations is listed below.

	Best-case	Worst-case
Searching for an integer in a list of integers	1	N
Searching a list of integers to make sure an integer not exists	1	N
Searching for an integer K in a list of integers where K is known to exist	1	N-1

Worst-Case Analysis of Sequential Search

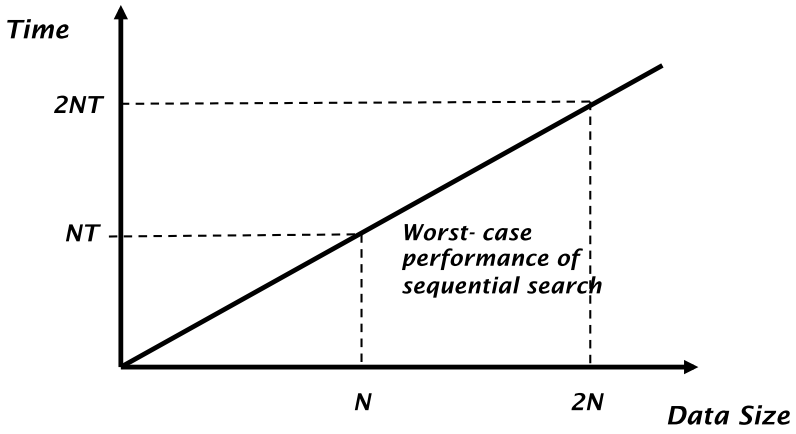
This section describes a worst-case analysis of sequential search.

Recall that sequential search involves checking or examining each data one by one. If the list has N elements, then in the worst case, N checking is required.



The time taken can be estimated based on the assumption that each comparison takes time T.

- N elements, the total time taken would be N times T (equals NT).
- 2N elements, the total time taken would be 2N times T (equals 2NT).



The sequential search algorithm **scales linearly** with the data size. This is an acceptable performance characteristic. Remind that this is the worst case.

Average-Case Analysis of Sequential Search

The average-case analysis is based on this assumption:

- The probability of finding a target in each element of a list is the same.
- The average time is the average of the time taken if the traversal ends at index 0, …, to index N-1.

Consider a list of N elements and we wish to look for a target K.

- Assume that target K must exist in the array, and so it can be found in any one of the n elements.
- The target may exist in the first element (the best case), the second element, and so on, and it may exist in the last element (the worst case).
- There are N possible situations.
- The number of element comparisons (and therefore proportionally the time taken) for each situation is listed in the following table.

Situations	***Number of Element Comparison***
K at element 1 (best case)	1
K at element 2	2
K at element 3	3
...	...
K at element n	N

Assume that there is equal chance that the target may be found in each of the N elements, and so the N possible situations may occur in equal chance. The average number of element comparison.

(1 + 2 + 3 + ... + N) divided by N

The series of addition (1 + 2 + 3 + ... + N) is equivalent to (1 + N) N / 2.

Therefore the average number is (1 + N) / 2.

If time taken for each element examination is T, the average time taken for searching an array of n elements is (1 + N) T / 2 or 0.5 (1 + N) T

- If N is large, then 0.5 (1 + N) T is approximately 0.5 NT.
- The average time taken of sequential search is directly proportional to N.
- Recall that the worst-case performance is NT.
- Recall that the best-case performance is 1.

The worst-case and the average-case have the same linear scalability, which means that sequential search scales well.

5.2.4 Efficiency Evaluation of Sequential Search: Empirical Approach

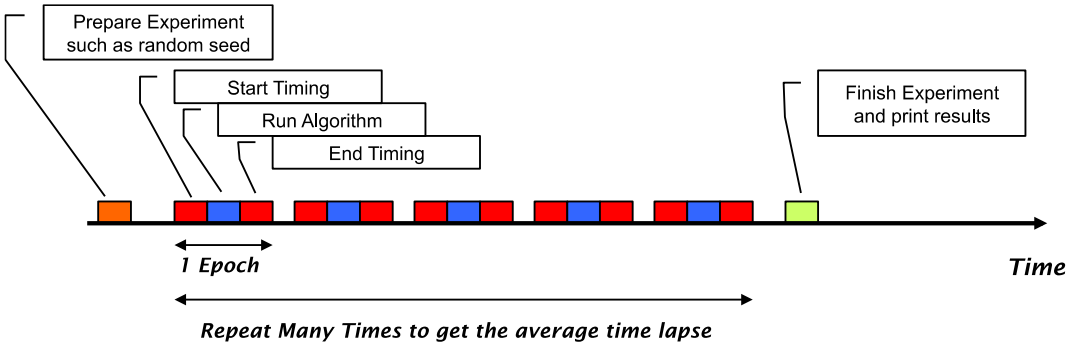
The **empirical approach** estimates the performance of a method or an algorithm by carrying experiments to measure the performance. In measuring the efficiency of sequential search, the empirical approach would measures the time taken of a searching operations given different data sizes.

- The absolute time taken is not important.
- The absolute time taken is clearly dependent on the power of the host computer.
- The performance characteristic of interest is the scalability of sequential search: the increased time taken given an increased data size.

Experimental Design

There are a few principles to setup an experiment so that the empirical performance can be measured correctly.

- The scalability is measured with different data sizes. For example, the time taken when data sizes is N, 2N, 4N, and 8N, etc.
- For each data size, multiple runs should be carried out. The time taken for all the runs should be averaged. Each run is called an **epoch**.
- For each epoch, the dataset should be randomized so that the likelihood of best-case and worst-case can also be averaged-out.



❖ *Taking Time in Python*

Python provides the `time` module for getting information about the time.

Method	Remarks
`time.time()`	Return the Unix epoch time, which represents the number of seconds since Jan 1, 1970 00:00:00

Interactive Mode

```
>>> time.time()
1545714501.87726
```

Experiment 1: Time Taken for Worst-Case Sequential Search

The following program generates a random list of N integers and then search for the maximum in the list.

- Searching the maximum in an unsorted list needs to traverse the whole list.
- It measures the worst-case performance of sequential search.

There are at least a couple of ways to force a worst-case performance in sequential search

- Finding maximum or minimum, which ensures traversing the whole list.
- Searching for a number that does not exist in the list, which again ensures traversing the whole list.

Example: linear-search-exp-1.py

```
import random
import time

def genlist(size):
    return random.sample(range(0, size), size)

if __name__ == "__main__":
    # experimental parameters
    dataSize = 10000
    epoch = 1000

    random.seed(1)
    # generate dataset
    numlist = genlist(dataSize)
    # start timing
    startTime = time.time()
    for e in range(0, epoch):
        max(numlist)
    endTime = time.time()

    # caclulate time taken
    timeTaken = (endTime - startTime) / epoch
    # convert to milli-seconds
    timeTaken *= 1000
    print("Time taken is {} ms for data size {}".format(timeTaken, dataSize))
```

```
Time taken is 0.00012599945068359375 seconds for data size 10000
```

Notes:

- The program allows changing of experimental parameters such as data size and number of epoch easily.
- The hardcoded random seed of 1 allows the experiment to be repeated later with exactly the same condition.
- The epoch is set to 1000. It means running the experiment 1000 times and then getting the average.

The results of repeatedly running the program with different data sizes are shown below.

```
Time taken is 0.056192874908447266 ms for data size 5000
Time taken is 0.12858295440673828 ms for data size 10000
Time taken is 0.2781240940093994 ms for data size 20000
Time taken is 0.6104047298431396 ms for data size 40000
```

The results are tabulated below

Size	***Time Taken (milliseconds)***
5000	0.0561
10000	0.129
20000	0.278
40000	0.610

Experiment 2: Running Different Data Sizes and Graphing the Results

The following program has improved upon the previous version.

- It has modularized the experiment part into a function for repeated calling.
- It tries different data size.
- It graphs the results for better visualization.

Example: linear-search-exp-2.py

```
import random
import time
import matplotlib.pyplot as plt
```

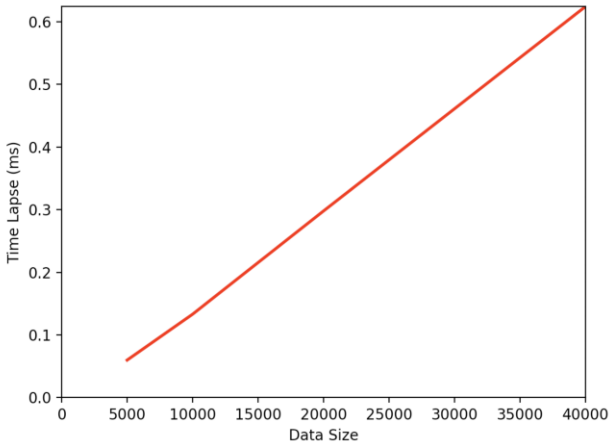
```
def genlist(size):
    return random.sample(range(0, size), size)
```

```
# this function returns the time taken in ms
# for worst case sequential search
def searchlist(numlist, epoch):
      # start timing
    startTime = time.time()
    for e in range(0, epoch):
        max(numlist)
    endTime = time.time()
    # caclulate time taken
    timeTaken = (endTime - startTime) / epoch
    # convert to milli-seconds
    timeTaken *= 1000
    return timeTaken
```

```
if __name__ == "__main__":
    # experimental parameters
    dataSizeList = [5000, 10000, 20000, 40000]
    epoch = 1000
    # experimental results
    timeTakenResults = list()

    random.seed(1)

    for dataSize in dataSizeList:
        # generate dataset
        numlist = genlist(dataSize)
        timeTaken = searchlist(numlist, epoch)
        print("Time taken is {} ms for data size {}".format(timeTaken, dataSize))
        timeTakenResults.append(timeTaken)


    plt.xlabel('Data Size')
    plt.ylabel('Time Lapse (ms)')
    plt.axis([0, dataSizeList[-1], 0, timeTakenResults[-1]])
    plt.plot(dataSizeList, timeTakenResults, color='r', linewidth=2.0)
    plt.show()
```

```
Time taken is 0.05977916717529297 ms for data size 5000
Time taken is 0.13311004638671875 ms for data size 10000
Time taken is 0.2979559898376465 ms for data size 20000
Time taken is 0.6243178844451904 ms for data size 40000
```

The program generated the following output graph on my computer.



Notes:

- The graph clearly shows a linear scalability, which is the performance characteristic of worst-case sequential search.

Experiment 3: Time Taken for Average-Case Sequential Search

The average-case results can be obtained by searching for a random data that is known to exist in the list.

- The data can be located anywhere in the list.
- In a different epoch, the data should locate in a different place.

The previous program is changed so that the sequential search for a target known to exist in the list.

Example: linear-search-exp-3.py

```
# this function returns the time taken in ms
# for average case sequential search
def searchlist(numlist, epoch, size):
    # start timing
    startTime = time.time()

    for e in range(0, epoch):
        target = random.randrange(0, size) # the target must exist in the list
        for i in numlist:
            if i == target:  # if the target is found, stop looping
                break
        else:
            print("Target not found")

    endTime = time.time()
    # caclulate time taken
    timeTaken = (endTime - startTime) / epoch
    # convert to milli-seconds
    timeTaken *= 1000
    return timeTaken
```

```
...

    for dataSize in dataSizeList:
        # generate dataset
        numlist = genlist(dataSize)
        timeTaken = searchlist(numlist, epoch, dataSize)
        print("Time taken is {} ms for data size {}".format(timeTaken, dataSize))
```

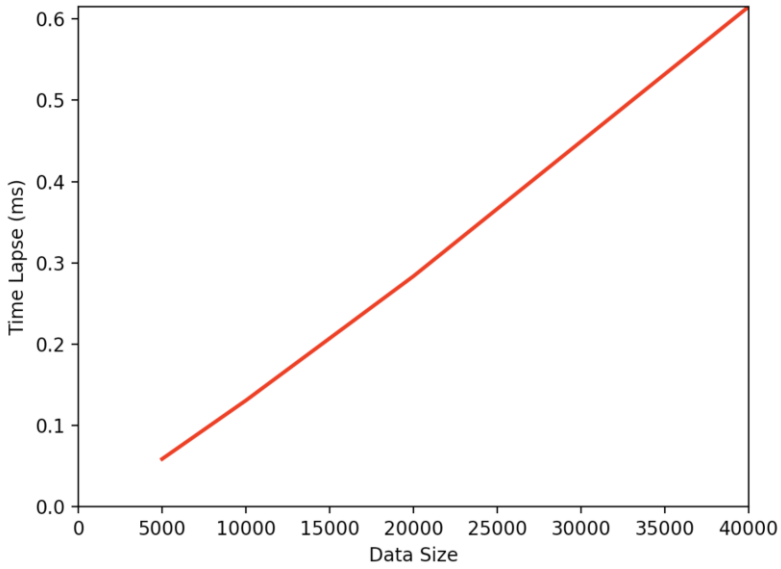
```
        timeTakenResults.append(timeTaken)
...
```

Notes:

- The bold part in the function searchlist has been changed to searching for a target known to exist in the list.
- The list contains integers ranged from 0 to dataSize – 1, but in a random order.
- The target is generated randomly from the range 0 to dataSize – 1.
- The else part of the inner for loop is never run during the experiment, ensuring that the target really always exists in the list.

The results are very similar to the worst-case experiment.

```
Time taken is 0.05881500244140625 ms for data size 5000
Time taken is 0.13083410263061523 ms for data size 10000
Time taken is 0.28359103202819824 ms for data size 20000
Time taken is 0.6148819923400879 ms for data size 40000
```



5.3 Efficient Searching Algorithm: Binary Search

Recall that a key principle of data searching is that proper data organization can speed up data searching. One very useful way to organize data is to have the data sorted in order.

Consider the following analysis.

	Best-case	Worst-case
Searching for the maximum in an unsorted list of integers	N	N
Searching for the maximum in a sorted list of integers	1	1
Searching for an integer K in an unsorted list of integers where K is known to exist	1	N-1
Searching for an integer K in an sorted list of integers where K is known to exist	?	?

Notes:

- The maximum in a sorted list of integers must be at the end of the list. So the number of checking or examination needed is 1.

- The maximum of an unsorted list of integers is only known if the whole list is examined.

There is one unknown in the analysis: can a sorted list facilitate efficient searching?

The answer is Yes, but not with sequential search. A better algorithm called binary search can exploit the order in sorted lists and result in very efficient search.

5.3.1 Binary Search

Sorted data in a list (or arrays) has useful characteristics:

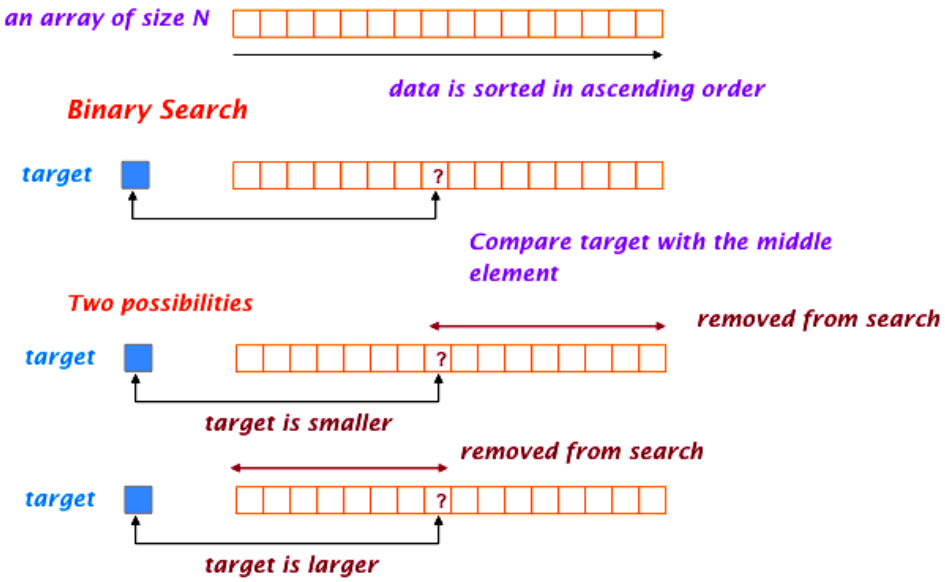
- For an array element, all data to the left are smaller.
- For an array element, all data to the right are larger.

The operation of binary search is described below.

Method of Binary Searching on an array

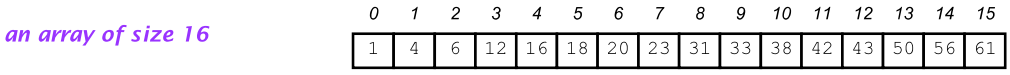
```
…
Given an array of size N
Define the search space of the array with two pointers: lower and upper.
Set lower = 0; upper = N-1
Loop (while there is data record in the search space) {
        Find the middle element from lower and upper
        Perform computation relevant to the data searching operation (such as comparison) on the middle element
        If Found, break;
        If the middle element is too small, narrow the search space by setting lower to middle+1
        If the middle element is too large, narrow the search space by setting upper to middle-1
}
...
```

The following figure describes one step of binary search.

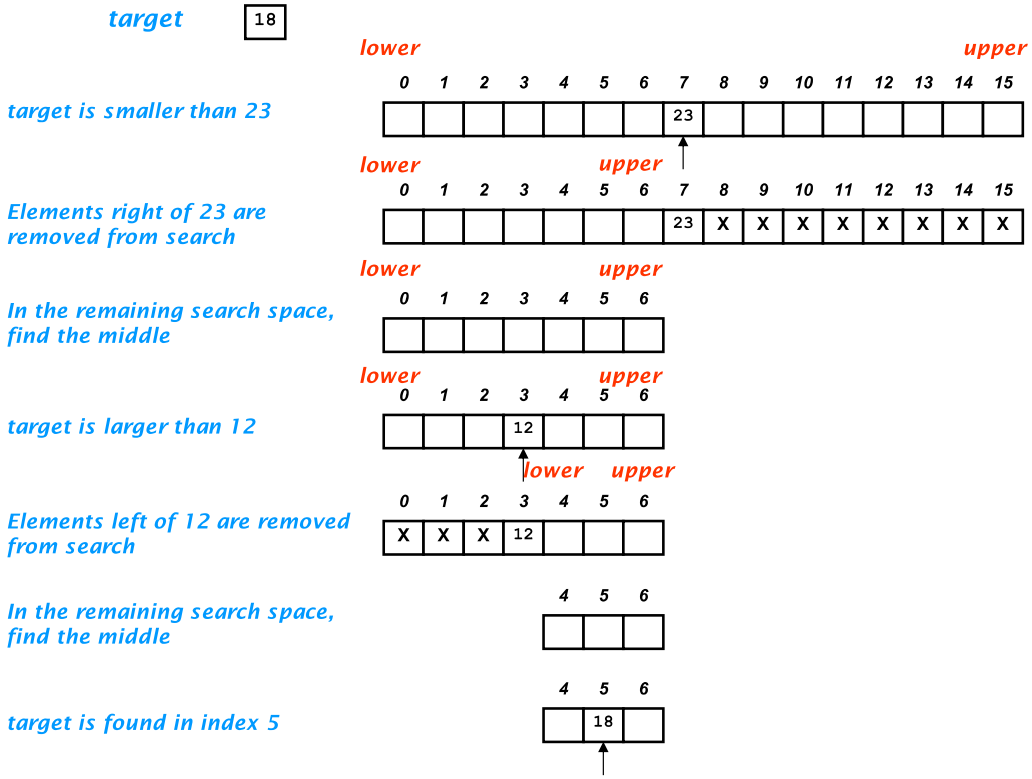


⊙ *Worked Example*

Given the following sorted list, use binary search to look for the value of 18.



The steps are shown below.



Notes:

1. Binary search works by always examine the middle element in the array.
2. Because the above array has even number of elements, there are two elements that could be considered as the middle. They are index 7 or index 8 respectively. We use index 7 (the smaller one) as the middle element.
3. We find that index 7 has value 23. We compare it with the target (18).
4. There are three possibilities at this point.
    - The element being examined matches the target. The target is found and searching completes.
    - The element being examined is greater than the target. Further searching can focus on smaller elements (elements smaller than the current element).
    - The element being examined is smaller than the target. Further searching can focus on larger elements (elements larger than the current element).

The following shows an implementation of binary search as a function.

Example: binary-search.py

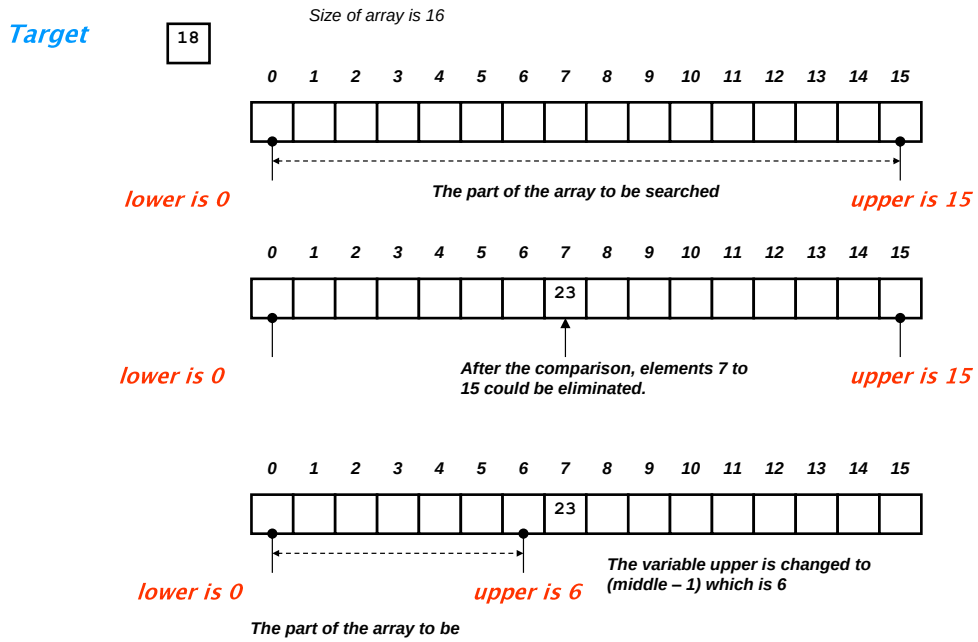
```
import random
import time
import matplotlib.pyplot as plt
```

```
def genlist(size):
    # generate numbers in the list in the range 10 times the size
    # if the size is 1000, it select 1000 integers from 0 to 10000 - 1
    return random.sample(range(0, size * 10), size)
```

```
# this function returns the index where the target is found in the list
# returns None if the target is not found
# numlist MUST BE SORTED

def binarySearch(numlist, target):
    size = len(numlist)
    lower = 0
    upper = size - 1
    while lower <= upper:
        middle = (lower + upper) // 2
        if target == numlist[middle]:
            return middle
        elif target > numlist[middle]:
            lower = middle + 1
        else:
            upper = middle - 1
    return None
```

Notes:

- The variables `lower` and `upper` are indexes of the list that keep track of which part of the list or array is to be searched.
- Binary search involves eliminating a whole part of elements after each examination.
- The variables are defined to keep the boundaries of the array part to be searched. It is illustrated in the following figure.



The following main program shows how to use the binary search. Most importantly, the list must be sorted so that binary search can be used.

```
if __name__ == "__main__":
    dataSize = 100
    numlist = genlist(dataSize)

    # sort the list
    numlist.sort()

    # draw a random nunmber from 0 to dataSize-1 as the target
    target = random.randrange(0, dataSize)
    index = binarySearch(numlist, target)
    print("The target {} is found at index {} in the list".format(target, index))

    # draw a nunmber randomly from numlist as the target
    target = random.choice(numlist)
    index = binarySearch(numlist, target)
    print("The target {} is found at index {} in the list".format(target, index))
```

```
The target 66 is found at index None in the list
The target 119 is found at index 13 in the list
```

Notes:

- Two examples are given above
- The first example draws a random number in the range from 0 to dataSize – 1 as target. The target may not exist in the list. So the function binarySearch may return None in such case.
- The second example draws a number randomly from the list as target. The target is known to exist in the list.

5.3.2 Performance of Binary Search: Analytical Approach

This section discusses worst-case analysis of binary search. The worst-case scenario happens in binary search if the target does not exist in the array (the same case as in sequential search).

- A characteristic of binary search is that after each mismatched comparison, either the lower half or the upper half of the current elements in the array is eliminated from further searching.
- For example, there are currently n elements being searched. After a mismatched comparison, the remaining elements should be (n/2) or (n/2)-1, depending on the oddity of the array.
- Given that the worst-case scenario is considered, (n/2) is chosen as the number of remaining elements.

The following table shows the number of remaining elements as a binary search progresses. The number of elements is 100000 to start with.

Examinations	***Number of Elements in the Searching Part***	***Examinations***	***Number of Elements in the Searching Part***
-	100000	9	195
1	50000	10	97
2	25000	11	48
3	12500	12	24
4	6250	13	12
5	3125	14	6
6	1562	15	3
7	781	16	1
8	390	17	0

The following works out the worst-case number of examinations required in binary search.

Derivation

The general formula for finding the worst-case number of element examinations for binary search is given in the following.

Number of Element Examinations = $\log_{\text{base } 2} N$

It is more commonly expressed as the following.

Number of Element Examinations = $\lg N$

The symbol n is the number of elements to be searched. The symbol lg means $\log_{\text{base } 2}$. Most calculators do not provide a log base 2 function. The following formula is more useful with calculators.

Number of Element Examinations = $(\log_{\text{base } 10} N) / (\log_{\text{base } 10} 2)$

If N is 100000, the above formula gives 16.61.

The following shows how the formula is derived. Given that there are N data, and K number of comparisons is required in the worst-case scenario.

# Examinations	***Data Size***		***Remarks***
0	N	$N \times 2^0$	
1	N // 2	$N \times 2^{-1}$	
2	N // 4	$N \times 2^{-2}$	
...			
K – 1	1	$N \times 2^{-(K-1)}$	$1 = N \times 2^{-(K-1)}$
K	0	$N \times 2^{-K}$	

A formula linking K and N is found in the above table.

Derivation

$1 = N \times 2^{-(K-1)}$

$2^{(K-1)} = N$

$(K - 1) = \log_2 N$

$K \sim \log_2 N$

Performance of Binary Search: Sub-linear Scalability

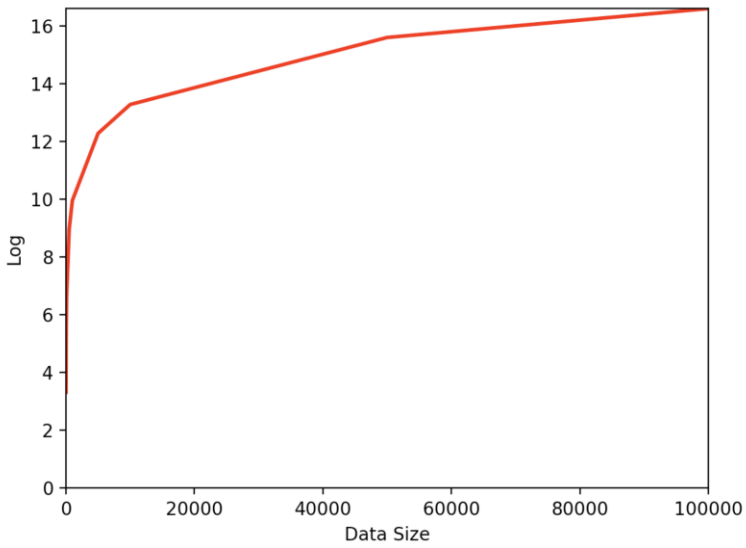
The following table compares the number of element examinations for different data sizes using binary search and sequential search. The high performance of binary search is revealed when the data size is large.

Number of Elements	*Binary Search (Worst-Case)*	*Sequential Search (Worst-Case)*
10	4	10
100	7	100
1000	10	1000
10000	14	10000
100000	16	100000

The shape of log base 2 is best shown in a graph using a program.

Example: show-log.py

```
import math
import matplotlib.pyplot as plt

xlist = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000]
ylist = [math.log2(x) for x in xlist]


plt.xlabel('Data Size')
plt.ylabel('Log2')
plt.axis([0, xlist[-1], 0, ylist[-1]])
plt.plot(xlist, ylist, color='r', linewidth=2.0)
plt.show()
```



The graph of log base 2 clearly shows a sub-linear scalability characteristic.

5.3.3 Performance of Binary Search: Empirical Approach

This section describes the use of empirical approach to confirm that performance analysis that binary search shows a log performance characteristic. This performance characteristic is sub-linear scalability.

Experimental Design

The experimental design is similar to that for the sequential search analysis. There is one important different. The dataset in the list is sorted before applying binary search.

Experiment: Time Taken for Worst-Case Binary Search

The worst-case results can be obtained by searching for a non-existent target.

- All random data in the list are in the range 0 to dataSize – 1. It means all positive integers.
- So a negative integer can be used as an non-existent target.

The implementation of the binary search is the same as the previous section.

Example: binary-search-exp-1.py

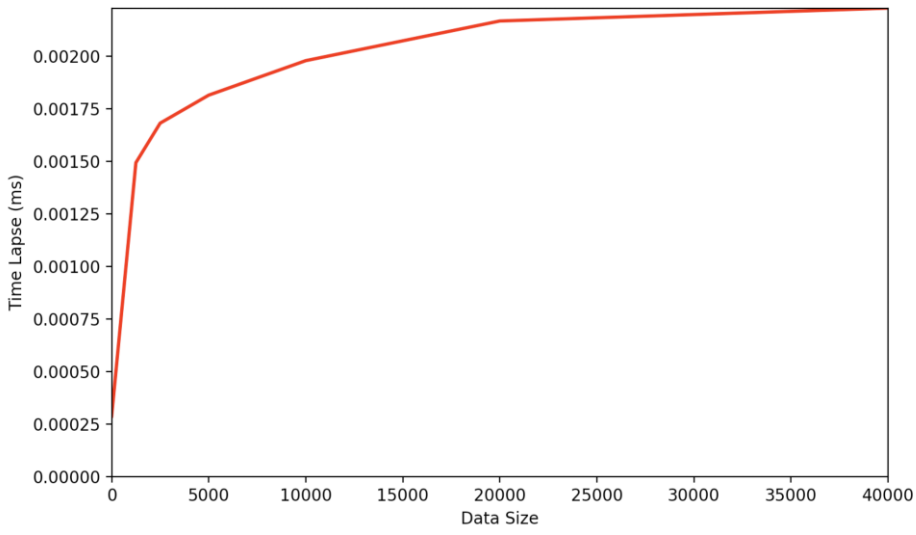
```
import random
import time
import matplotlib.pyplot as plt

def genlist(size):
    # generate numbers in the list in the range 10 times the size
    # if the size is 1000, it select 1000 integers from 0 to 10000 - 1
    return random.sample(range(0, size * 10), size)

# this function returns the index where the target is found in the list
# returns None if the target is not found
# numlist MUST BE SORTED
def binarySearch(numlist, target):
    size = len(numlist)
    lower = 0
    upper = size - 1
    while lower <= upper:
        middle = (lower + upper) // 2
        if target == numlist[middle]:
            return middle
        elif target > numlist[middle]:
            lower = middle + 1
        else:
            upper = middle - 1
    return None
```

```
# this function returns the time taken in ms
# for worst case binary search
def searchlist(numlist, epoch, size):
    # start timing
    startTime = time.time()

    for e in range(0, epoch):
        # target of -1 forces worst case
        # because -1 does not exist in numlist
        index = binarySearch(numlist, -1)

    endTime = time.time()
    # caclulate time taken
    timeTaken = (endTime - startTime) / epoch
    # convert to milli-seconds
    timeTaken *= 1000
    return timeTaken
```

```
if __name__ == "__main__":

    # experimental parameters
    dataSizeList = [1, 1250, 2500, 5000, 10000, 20000, 40000]
    epoch = 1000
    # experimental results
    timeTakenResults = list()

    random.seed(1)

    for dataSize in dataSizeList:
        # generate dataset and sort the list
        numlist = genlist(dataSize)
        numlist.sort()

        timeTaken = searchlist(numlist, epoch, dataSize)
        print("Time taken is {} ms for data size {}".format(timeTaken, dataSize))
        timeTakenResults.append(timeTaken)


    plt.xlabel('Data Size')
    plt.ylabel('Time Lapse (ms)')
    plt.axis([0, dataSizeList[-1], 0, timeTakenResults[-1]])
    plt.plot(dataSizeList, timeTakenResults, color='r', linewidth=2.0)
    plt.show()
```

The following shows the results in the output and the graph.

```
Time taken is 0.00028705596923828125 ms for data size 1
Time taken is 0.0014929771423339844 ms for data size 1250
Time taken is 0.0016808509826660156 ms for data size 2500
Time taken is 0.0018141269683837891 ms for data size 5000
Time taken is 0.0019978158950805664 ms for data size 10000
Time taken is 0.0021669864654541016 ms for data size 20000
Time taken is 0.0022280216217041016 ms for data size 40000
```



Notes:

- The graph shows a characteristic very similar to the log graph.
- The performance characteristic is sub-linear.

COMPS209F Data Structures, Algorithms and Problem Solving

Sorting Algorithms 6

A sorted list of data can facilitate efficient data operations such as data searching. However, a sorting operation must be carried out first if a list of data is not already sorted.

This chapter will cover the major sorting algorithms. Sorting is a time consuming operation. The performance of the sorting algorithms will be analysed. However, the chapter will first describe the functions provided by Python for sorting a list.

6.1 Sorting Functions in Python

Python provides the following built-in function for sorting a list.

```
sortedlist = sorted(list, key=..., reverse=...)
```

- The key and the reverse named parameters are optional.
- The `key` parameter is a function providing the key for sorting order and comparison. It is not required if the elements in the list has a natural ordering (such as numbers)
- The `reverse` parameter has default value of `False`. If it is `True`, the sorting will be in descending order.
- The function returns a new list that is sorted.

Alternatively, the method sort() of the list object will do the same thing.

```
list.sort(key=..., reverse=...)
```

Example 1: Sorting Numbers

The following program shows different ways to sort a list of numbers.

Example: sort-numbers.py

```
# using the function sorted
# the function returns the sorted list
listA = [10, 5, 2, 8, 3, 4, 9, 1]
newlist = sorted(listA)
print(newlist)                                  [1, 2, 3, 4, 5, 8, 9, 10]

# using the function sorted
# the function returns the sorted list
listA = [10, 5, 2, 8, 3, 4, 9, 1]
newlist = sorted(listA, reverse=True)
print(newlist)                                  [10, 9, 8, 5, 4, 3, 2, 1]

# using the sort() method of list object directly
# NOTE: the method directly sorts the list object
listA = [10, 5, 2, 8, 3, 4, 9, 1]
listA.sort()
print(listA)                                    [1, 2, 3, 4, 5, 8, 9, 10]
```

```
# strings are NOT sort-able
# strings can be converted to list first
# the characters in the list are then sorted
str = "ANDERS CHAN"
listB = list(str)
listB.sort()
print(listB)
```

```
[' ', 'A', 'A', 'C', 'D', 'E', 'H', 'N', 'N', 'R', 'S']
```

Example 2: Using Key to Sort a List of Dictionaries

A list of dictionaries has multiple fields in its elements. Such lists are not sortable unless a key is provided for the elements.

The following example shows the same dataset containing population information of major US cities. There are three fields in the dictionaries:

- `Name` (the name of a city)
- `State` (the state where the city is located)
- `Pop` (the population of the city as an integer)

To sort this list, the key parameter must be provided. The following program shows how to define a function as the key parameter.

- The only parameter of these key functions is an element in the list.
- The function should return a value that can be sortable (i.e. has a natural order such as numbers or strings).

Example: sort-with-key-2.py

```
def getName(city):
    return city['Name']

def getPop(city):
    return city['Pop']
```

```
cities = list()
isFirstLine = True
try:
    with open('us-cities.csv','r') as infile:
        while True:
            line = infile.readline()
            if line:
                if isFirstLine:    # skip the first line
                    isFirstLine = False
                    continue
                row = line.rstrip().split(',') # apply rstrip and then split into a list
                city = dict()  # create a dictionary for one city
                city['Name'] = row[0]
                city['State'] = row[1]
                city['Pop'] = int(row[2])
                cities.append(city)
            else:
                break
except IOError:
    print("Error in opening file")
    sys.exit()
```

```
# sort ascending by 'Name'
cities.sort(key=getName)
print(cities)

# sort descending by 'Pop'
cities.sort(key=getPop, reverse=True)
print(cities)
```

Example 3: Define Key with Multiple Attributes to Sort a List of Dictionaries

Sometimes sorting is based on multiple attributes. For example, one may want the elements sorted by the State, following by the Name, and lastly by the Population.

This can be achieved by the key function defining a returning a tuple based on desired order of fields.

Example: sort-with-key-3.py

```
def multipleKey(city):
    return (city['State'], city['Name'], city['Pop'])
```

```
...
# sort ascending by 'State', 'Name' and then 'Pop'
cities.sort(key=multipleKey)

for city in cities:
    print(city['State'], city['Name'], city['Pop'])
```

```
Alabama Birmingham 212038
Alabama Huntsville 183739
Alabama Mobile 194822
Alabama Montgomery 205293
Alaska Anchorage 298610
Arizona Chandler 245628
Arizona Gilbert 221140
...
```

- The function returns a tuple that makes up of three items in the order of the ‘State’ field, the ‘Name’ field, and the ‘Pop’ field of an element.

Example 4: Define Key with Lambda Functions

Instead of defining a named function, one can use lambda functions for convenience.

Example: sort-with-key-4.py

```
# sort descending by 'Pop'
cities.sort(key=lambda city: city['Pop'], reverse=True)
```

```
# sort ascending by 'State', 'Name' and then 'Pop'
cities.sort(key=lambda city: (city['State'], city['Name'], city['Pop']))
```

Notes:

- The first example sorts the list according to the population in descending order.
- The second example sorts the list by the state, following by the name, and lastly by the population.

Example 5: Define Key with operator.itemgetter()

The module operator provides a function itemgetter() for defining key conveniently. The following examples show how to carry out the same sorting with itemgetter().

Example: sort-with-key-5.py

```
import operator

# sort descending by 'Pop'
cities.sort(key=operator.itemgetter('Pop'), reverse=True)
```

```
import operator

# sort ascending by 'State', 'Name' and then 'Pop'
cities.sort(key=operator.itemgetter('State', 'Name', 'Pop'))
```

Notes:

- Different experienced python programmers thinks the itemgetter() function differently. Some consider it better than lambda functions but others think this is not as good.
- The itemgetter() function method is considered more expressive.
- The lambda function method is more general.

6.2 Principles of Sorting

There are a number of concepts related to sorting.

Ordering of Data

The sort order defines the manner how the data objects are arranged in a list. **Natural ordering** is found in numeric data and alphabets.

- Numbers: there is a greater-than and less-than relation between two numeric objects.
- Alphabets: there is a before and after relation between English alphabets.

Not all data types have a definite idea on the ordering. The following are some of these examples.

- Telephone numbers. There is no universal agreement on the ordering. Of course, numerically one telephone number can be compared to another telephone number (e.g. 12345678 is smaller than 99999998). Seldom would people compare and order telephone number numerically.
- OUID. Student identity number is an eight digit numeric data, which the first two digits represent the year first registered with the university.
- English words. There is a concept to sort it alphabetically (e.g. "orange" after "grapes"), or whether to consider "orange" and "Orange" the same rank. Should we instead arrange the English words according to the number of letters?

For data item without natural order, an order rule must be defined.

Ascending and Descending Order

Data can also be sorted in **ascending order** or **descending order**.

- Ascending order means data is arranged from low to high or from small to great.
- Descending order is the opposite.

Sorting Algorithms

A sorting algorithm is a method that arranges data according to a sort order.

There have been many sorting algorithms developed since the dawn of modern computing. Researchers wished to produce the best sorting algorithms but they found a strong trade-off characteristic in data sorting. A sorting algorithm is fast but it uses lots of memory. Another sorting algorithm is fast in average-case scenario but can be very poor in the worst-case scenario.

The following characteristics are often used to describe the performance of sorting algorithms (and other algorithms too).

- Best-case performance, average-case performance, and worst-case performance.
- Memory usage.
- Comparison complexity. It measures how complex is to work out the relative order of two data objects.
- Implementation complexity.

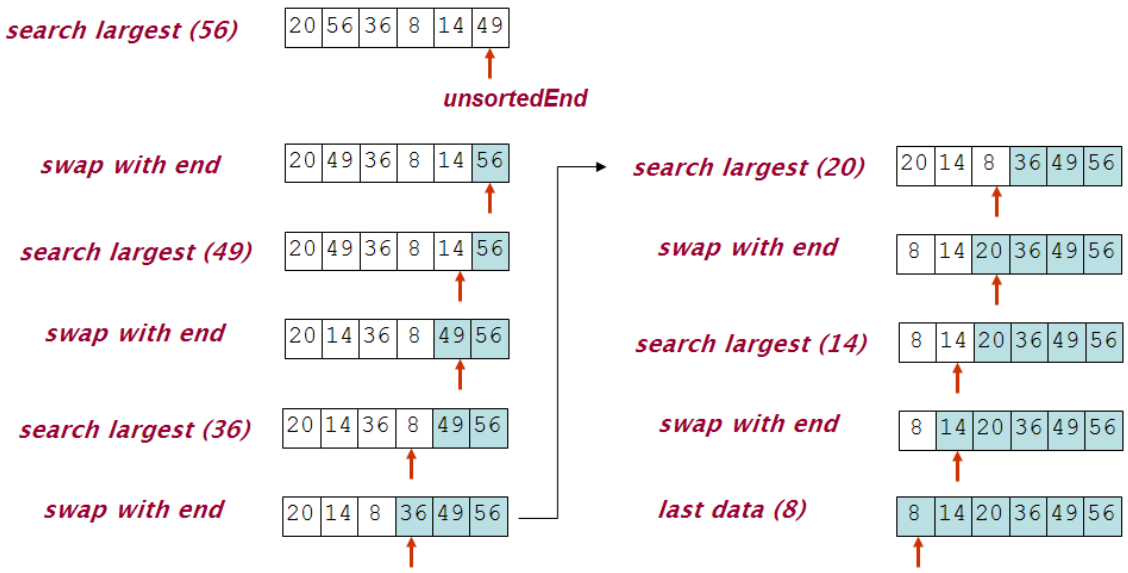
6.3 Selection Sort and Insertion Sort

Selection sort and Insertion sort are two basic types of sorting algorithms.

6.3.1 Selection Sort

Selection sort is succinctly described as search (largest) and swap. Selection sort searches for the largest (or smallest) data and moves it to the correct place by data swapping. The *selection* in the name refers to the searching operation.

Selection Sort (Ascending)



Notes:

- The key variable `unsortedEnd` that keeps track of the progress of the sorting operation.
- It separates the array into two parts: the sorted part (blue) and unsorted part (white).
- When `unsortedEnd` reaches index 0, the whole array is sorted.

The following shows an implementation of selection sort (in ascending order).

Example: selection-sort-1.py

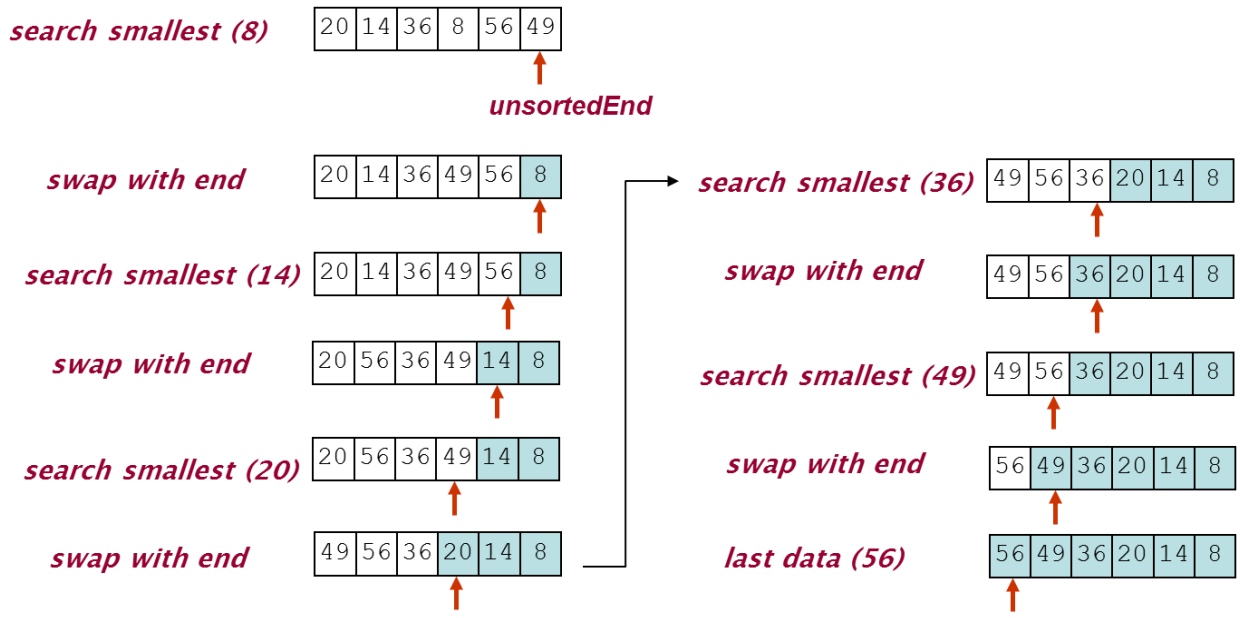
```
# Selection sort (Ascending order)
def sortSelectionAsc(alist):

    size = len(alist)
    # unsortedEnd runs from size-1 to 1
    for unsortedEnd in range(size - 1, 0, -1):
        # search for largest value in unsorted part
        lindex = 0    # index of largest value
        for i in range(1, unsortedEnd + 1):
            if alist[i] > alist[lindex]:
                lindex = i
        # swap largest with end
        alist[lindex], alist[unsortedEnd] = alist[unsortedEnd], alist[lindex]
        # print(alist)    # print intermediate steps
```

The intermediate steps printed (uncomment the print statement) are shown below.

```
[20, 49, 36, 8, 14, 56]
[20, 14, 36, 8, 49, 56]
[20, 14, 8, 36, 49, 56]
[8, 14, 20, 36, 49, 56]
[8, 14, 20, 36, 49, 56]
[8, 14, 20, 36, 49, 56]
```

Selection Sort (Descending)

There is one key difference: searching for the smallest instead of searching for the largest. The smallest values are swapped to the end during the process.



The following shows an implementation of selection sort (in descending order).

Example: selection-sort-1.py

```
# Selection sort (Descending order)
def sortSelectionDesc(alist):

    size = len(alist)
    # unsortedEnd runs from size-1 to 1
    for unsortedEnd in range(size - 1, 0, -1):
        # search for smallest value in unsorted part
        sindex = 0    # index of smallest value
        for i in range(1, unsortedEnd + 1):
            if alist[i] < alist[sindex]:
                sindex = i
        # swap smallest with end
        alist[sindex], alist[unsortedEnd] = alist[unsortedEnd], alist[sindex]
        # print(alist)    # print intermediate steps
```

6.3.2 Insertion Sort

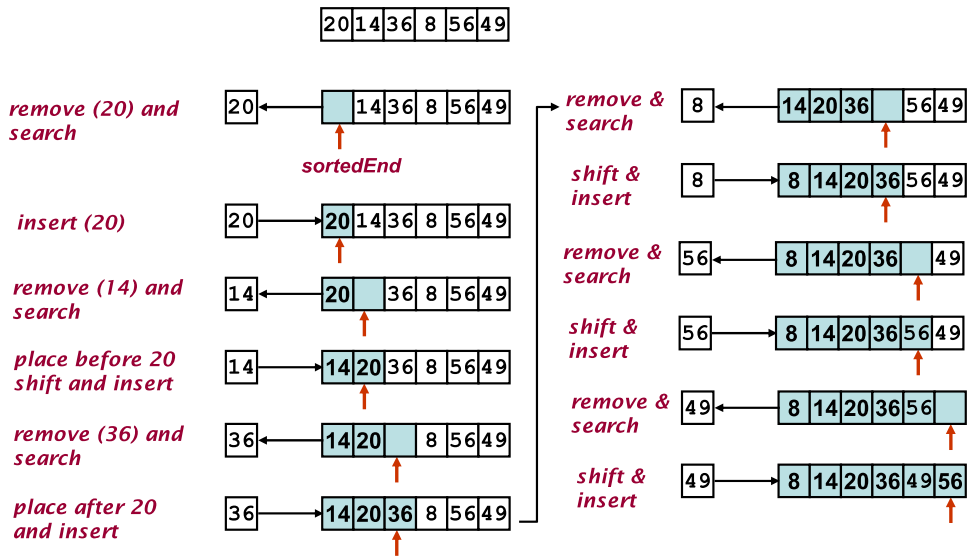
Insertion sort has two variants.

- Sorting an existing list or array.
- Adding new data to an existing sorted list.

Insertion Sort (Ascending) (Existing List)

Carrying out insertion sort in an existing array is succinctly described as remove, search, and insert. Insertion sort removes an unsorted data from an array to create a spare element. It searches for the right place to insert the data in the sorted part of the array. It then shifts data by making use of the spare element. Finally, the data is inserted at the place.

Insertion Sort (Ascending) (Existing Array)



Notes:

- There is a variable `sortedEnd` that keeps track of the progress of the sorting operation.
- It separates the array into two parts: the sorted part (blue) and unsorted part (white).
- When `sortedEnd` reaches the last index, the whole array is sorted.
- Insertion sort gradually creates and expands a sorted part in the array by repeated insertion.

The following shows an implementation of insertion sort on an integer list.

Example: insertion-sort-1.py

```
# Insertion sort (Ascending order)
def sortInsertionAsc(alist):

    size = len(alist)
    # sortedEnd runs from 0 to size-1
    for sortedEnd in range(0, size):
        toInsert = alist[sortedEnd]
        # find suitable place to insert
        insertLoc = 0
        while insertLoc < sortedEnd:
            if toInsert < alist[insertLoc]:
                break
            insertLoc += 1
        # insertLoc is the location to insert
        # right shift the remainng elements in the sorted part
        for i in range(sortedEnd, insertLoc, -1):
            alist[i] = alist[i-1]
        alist[insertLoc] = toInsert

        # print(alist)
```

Notes:

- A for structure controls the value of `sortedEnd` from 0 to (size – 1).
- Inside the outer for loop, there are three parts: remove & search, shift, and insert.

The intermediate steps printed (uncomment the print statement) are shown below.

```
[20, 14, 36, 8, 56, 49]
[14, 20, 36, 8, 56, 49]
[14, 20, 36, 8, 56, 49]
[8, 14, 20, 36, 56, 49]
[8, 14, 20, 36, 56, 49]
[8, 14, 20, 36, 49, 56]
```

Insertion Sort (Descending) (Existing List)

To insertion sort in a descending manner, one can simply change the suitable place to insert. The following shows an implementation of insertion sort on an integer list in descending manner.

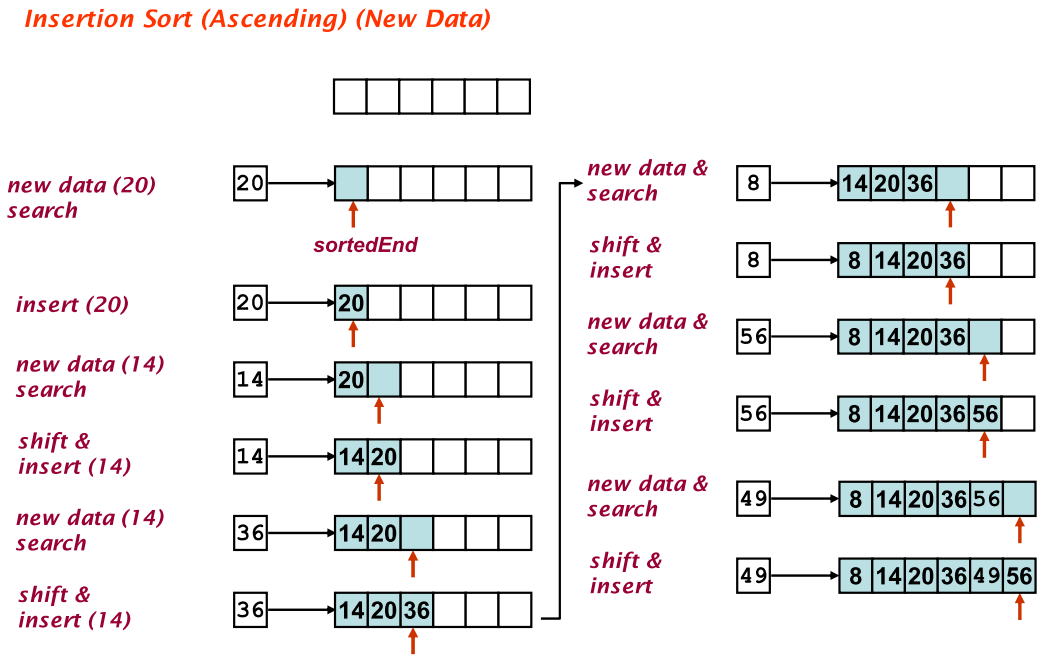
Example: insertion-sort-1.py

```
# Insertion sort (Descending order)
def sortInsertionDesc(alist):

    size = len(alist)
    # sortedEnd runs from 0 to size-1
    for sortedEnd in range(0, size):
        toInsert = alist[sortedEnd]
        # find suitable place to insert
        insertLoc = 0
        while insertLoc < sortedEnd:
            if toInsert > alist[insertLoc]:
                break
            insertLoc += 1
        # insertLoc is the location to insert
        # right shift the remainng elements in the sorted part
        for i in range(sortedEnd, insertLoc, -1):
            alist[i] = alist[i-1]
        alist[insertLoc] = toInsert

        # print(alist)
```

Insertion Sort (Ascending) (New Data)

The second variant of insertion sort is sorting data as new data is added. This is useful if the amount of data can change.

This operation mode is most suitable for insertion sort. There is no remove step in this operation mode. When a new data arrived, the algorithm will search for the right place in the sorted part of the array, shifts the data to create a space for insertion, and then insert the data.



The following shows an implementation of using insertion sort to add new data to a sorted list.

Example: insertion-sort-2.py

```
# Insertion sort (Ascending order) (new data)
def sortInsertionAsc(alist, toInsert):
    size = len(alist)
    insertLoc = 0

    # find suitable place to insert
    while insertLoc < size:
        if toInsert < alist[insertLoc]:
            break
        insertLoc += 1

    # insertLoc is the location to insert
    alist.insert(insertLoc, toInsert)
```

The following shows how to repeatedly call insertion sort to add a series of integers to a list. The list will remain sorted.

```
if __name__ == "__main__":
    numlist = list()

    sortInsertionAsc(numlist, 20)
    print(numlist)
    sortInsertionAsc(numlist, 14)
    print(numlist)
    sortInsertionAsc(numlist, 36)
    print(numlist)
    sortInsertionAsc(numlist, 8)
    print(numlist)
    sortInsertionAsc(numlist, 56)
    print(numlist)
    sortInsertionAsc(numlist, 49)
    print(numlist)
```

```
[20]
[14, 20]
[14, 20, 36]
[8, 14, 20, 36]
[8, 14, 20, 36, 56]
[8, 14, 20, 36, 49, 56]
```

The following shows another example that asks users to enter any number of integers. The integers are insertion sorted and stored in a list.

Example: insertion_sort_3.py

```
from insertion_sort_2 import sortInsertionAsc

# create an empty list
numlist = list()

while True:
    try:
        num = int(input("Enter an integer (Enter to stop): "))
        sortInsertionAsc(numlist, num)
        print(numlist)
    except ValueError:
        break

print("The final list:")
print(numlist)
```

```
Enter an integer (Enter to stop): 12
[12]
Enter an integer (Enter to stop): 20
[12, 20]
Enter an integer (Enter to stop): 8
[8, 12, 20]
Enter an integer (Enter to stop): 5
[5, 8, 12, 20]
Enter an integer (Enter to stop): 15
[5, 8, 12, 15, 20]
Enter an integer (Enter to stop): 6
[5, 6, 8, 12, 15, 20]
Enter an integer (Enter to stop):
The final list:
[5, 6, 8, 12, 15, 20]
```

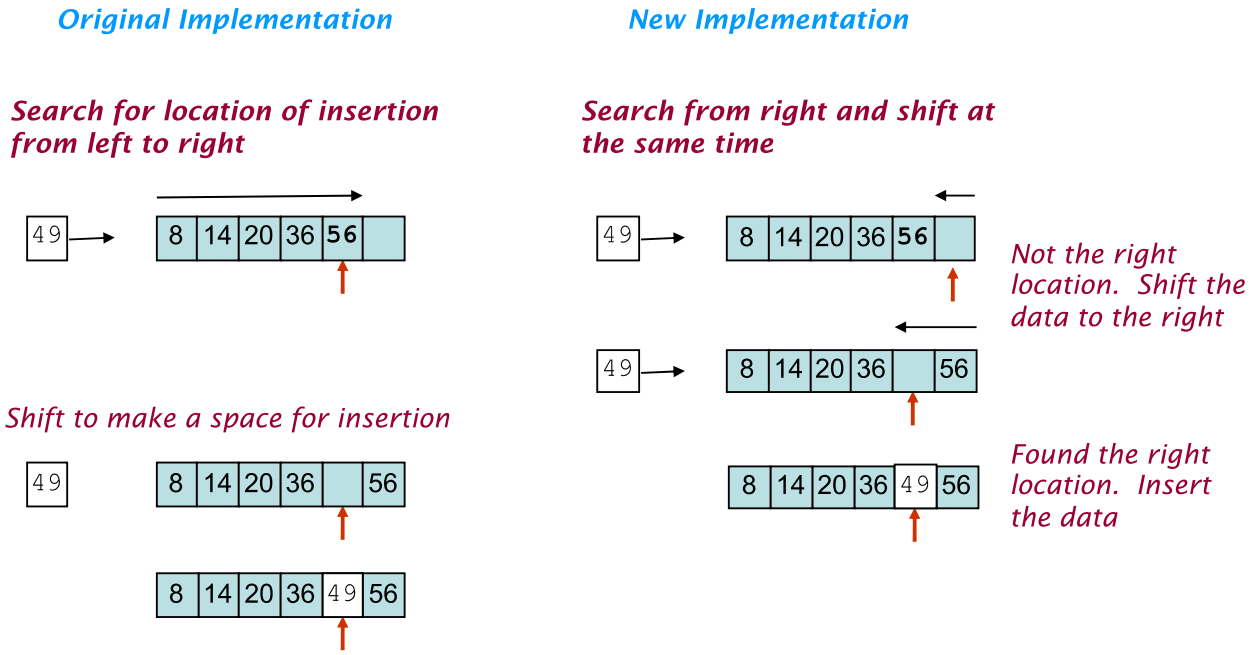
Notes:

- This Python program imports the function sortInsertionAsc() from insertion-sort-2.py.

Insertion Sort (Ascending) That Exploits Partially Sorted Lists (Optional)

A small modification to the insertion sort implementation can exploit a partially sorted list and finish sorting faster.

- For the search step, in the current implementation the search direction is from left to right. Shifting of data is carried out after the right location is found.
- In an improved implementation, the search direction should be from right to left and shifting data as the search if moving to the left.



The following shows the improved implementation of insertion sort.

Example: insertion-sort-4.py

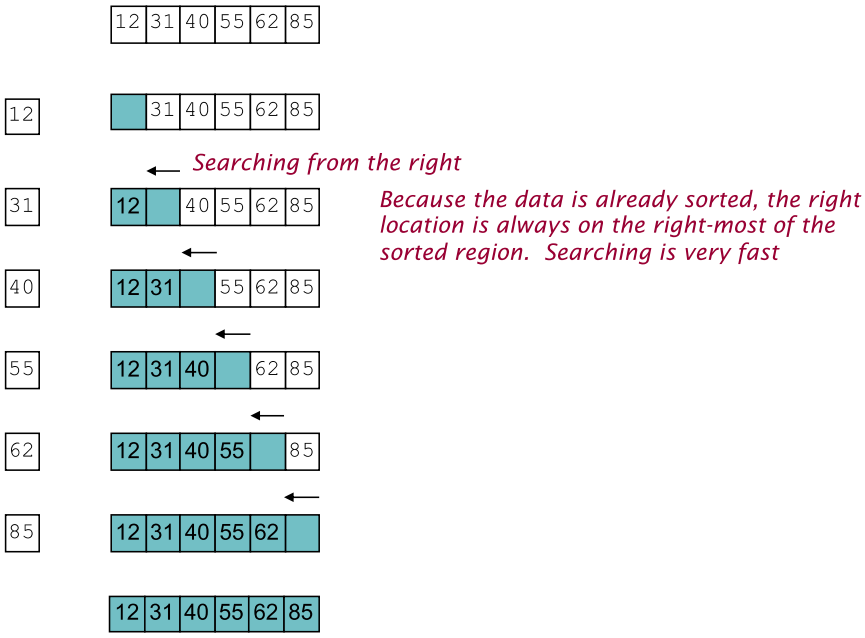
```
# Insertion sort (Ascending order)
def sortInsertionAsc(alist):

    size = len(alist)
    # sortedEnd runs from 0 to size-1
```

```
    for sortedEnd in range(0, size):
        toInsert = alist[sortedEnd]
        # find suitable place to insert
        insertLoc = sortedEnd
        while insertLoc > 0:
            if toInsert > alist[insertLoc - 1]:
                break
            # shift data and move to the left
            alist[insertLoc] = alist[insertLoc - 1]
            insertLoc -= 1
        alist[insertLoc] = toInsert

        #print(alist)
```

The following shows that for a sorted list, the search step will become very fast.

- The right location is always at the right most location of the sorted part.
- This is the first location checked by the search step.

Insertion Sort (Ascending) (Exploits Partially Sorted Lists)



6.3.3 Performance Analysis of Selection Sort and Insertion Sort: Empirical Approach

An experiment was carried out to investigate the scalability characteristic of selection sort and insertion sort. It was found that as the data size is doubled (2 times), the time is quadrupled (4 times).

Example: sort_analysis_1.py

```
from selection_sort_1 import sortSelectionAsc
from insertion_sort_1 import sortInsertionAsc
import random
import time
import matplotlib.pyplot as plt

def genlist(size):
    return random.sample(range(0, size * 10), size)

if __name__ == "__main__":
```

```
    # experimental parameters
    dataSizeList = [1000, 2000, 4000, 8000, 16000]
    epoch = 10
    # experimental results
    timeTakenResults = list()

    random.seed(1)
    for dataSize in dataSizeList:
        timeTaken = 0  # the total time for all epoches
        for e in range(0, epoch):
            # generate dataset
            numlist = genlist(dataSize)
            startTime = time.time()
            sortSelectionAsc(numlist)
            # sortInsertionAsc(numlist)
            endTime = time.time()
            timeTaken += (endTime - startTime)
        timeTaken /= epoch

        print("Time taken is {} s for data size {}".format(timeTaken, dataSize))
        timeTakenResults.append(timeTaken)


    plt.xlabel('Data Size')
    plt.ylabel('Time Lapse (s)')
    plt.axis([0, dataSizeList[-1], 0, timeTakenResults[-1]])
    plt.plot(dataSizeList, timeTakenResults, color='r', linewidth=2.0)
    plt.show()
```

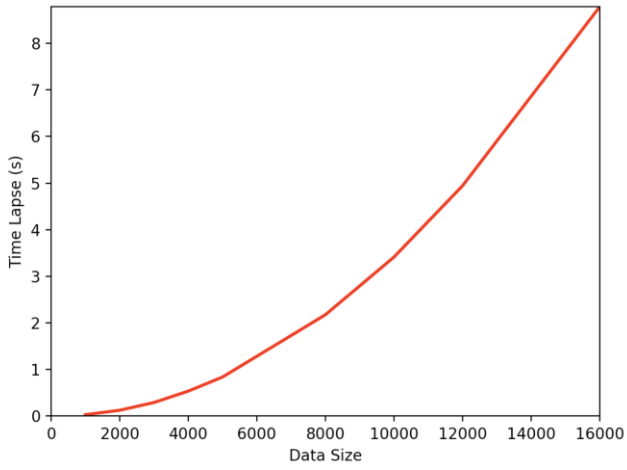
```
Time taken is 0.03019835948944092 s for data size 1000
Time taken is 0.12438504695892334 s for data size 2000
Time taken is 0.5191147089004516 s for data size 4000
Time taken is 2.1640388825035095 s for data size 8000
Time taken is 8.745663857460022 s for data size 16000
```

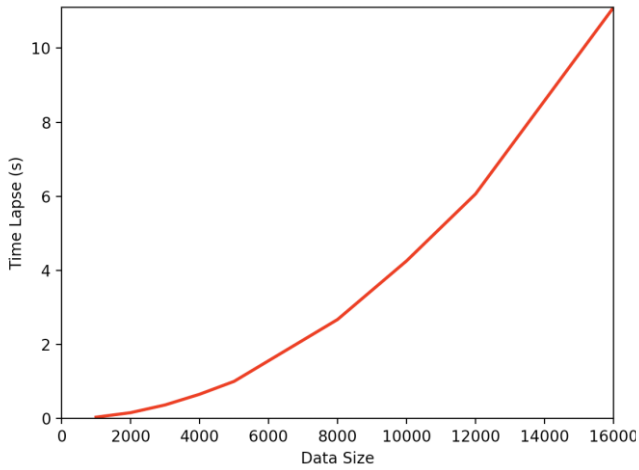
The absolute time taken is not an important indicator of the efficiency of an algorithm. The analysis revealed how the algorithms perform when they handle large data sizes.

Sorting Algorithm	***1000 elements (s)***	***2000 elements (s)***	***4000 elements (s)***	***8000 elements (s)***	***16000 elements (s)***
Selection Sort	0.0301	0.124	0.519	2.16	8.75
Insertion Sort	0.0357	0.149	0.633	2.53	10.35
Bubble Sort	0.0690	0.291	1.214	4.88	20.02

Notes:

- All the three sorting algorithms handle larger task size equally poorly.
- The trend can be seen graphically by plotting a graph. The graph below shows the performance characteristics of selection sort (left) and insertion sort. In fact, all the three algorithms would produce a similar graph.
- All three algorithms increase at a great rate with increasing data size.
- The increasing rate is super-linear.
- Doubling the data size will cause an approximately four times increase in time. The relation is quadratic.





If T is the variable for running time, and N is the data size, then the following ratio is found from the experiment.

T *proportional to* N^2

Alternatively,

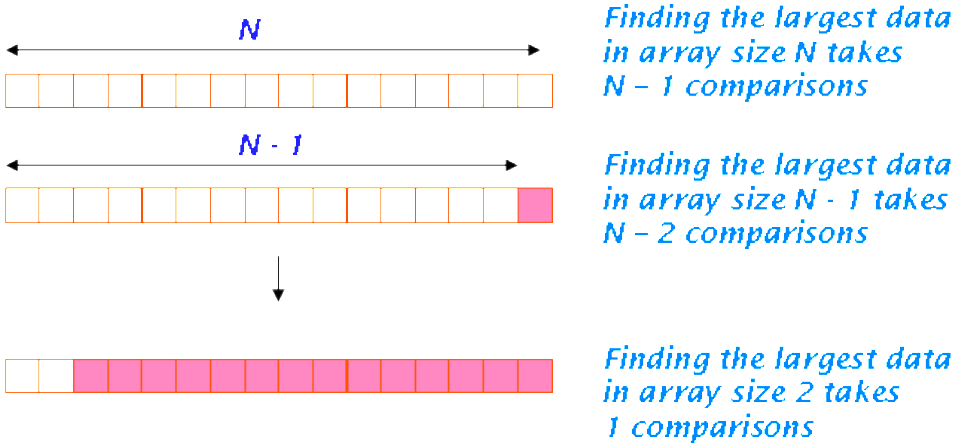
$T = c\ N^2$

6.3.4 Performance Analysis of Selection Sort: Analytical Approach

A similar relation is discovered from analytic approach of performance evaluation. The selection sort algorithm is analysed as follows.

The algorithm includes two operations: search for largest and swap. The swap operation is one-off for each data and it is not computationally complex. The search operation however can take long if there are a lot of data.

The following figure shows that in the worst-case, the searching operations contribute to the bulk of computation with their many data comparisons.



The total number of comparisons (or examinations) is the sum of each searching operation.

$Number\ of\ Comparisons = 1 + 2 + \ldots + (N-1) = N\,(N-1)\,/\,2$

$Number\ of\ Comparisons = 0.5\ N^2 - 0.5\ N$

When N is large, the fast increasing N2 term will make the N term insignificant.

$Number\ of\ Comparisons \approx 0.5\ N^2$ *(when N is large)*

Notes:

- The result of analytic evaluation is consistent with the result of empirical evaluation.
- The number of comparisons or examinations is always the same in these algorithms, regardless of the dataset.
- The best-case and the worst-case performance are therefore the same.

The following summarizes the performance of selection sort and insertion sort in terms of the number of examinations (or comparisons) with respect to data size N.

Sorting Algorithm	***Best-case***	***Average-case***	***Worst-case***
Selection Sort	N^2	N^2	N^2
Bubble Sort	N^2	N^2	N^2

6.4 Quick Sort

Hoare developed this efficient and elegant algorithm over 40 years ago.

In a nutshell, quicksort is based on a partition operation. Each partition operation on an array (or part of an array) involves selecting a pivot data and dividing other data objects into two sub-lists. The data in one sub-list are all smaller than the pivot data. The data in the other sub-list are all greater than the pivot data.

The partition operation is applied to the original list, and then the sub-lists, until all the sub-lists have only one value.

6.4.1 Operation of the Quicksort Algorithm

Quicksort is a recursive algorithm. It is applied to the original list to create sub-lists, and then applied to each of the sub-lists to create more sub-lists.

The following describes the algorithm in pseudo code.

Algorithm Quick Sort

```
Quicksort(list) {
    1   Partition the list into two sub-lists, left, and right, at a pivot element, where the left
        sub-list has elements smaller than the pivot element and the right sub-list has
        elements greater than the pivot element;
    2   Perform Quicksort on the left sub-list;
    3   Perform Quicksort on the right sub-list;
    4   Repeat until the list can no longer be divided.
}
Combine the sorted sub-lists.
```

Example of Executing Quicksort

Consider the following list.

(50 25 75 12 80 20 70 2 90 11 33 99 5 28 77 12)

Using 50 as the pivot, moving all elements smaller than 50 to its left and moving all elements larger than 50 to its right, we have:

(25 12 20 2 11 33 5 28 12) (**50**) (75 80 70 90 99 77)

There are three parts:

- A sub-list with all elements smaller than the pivot 50.
- A sub-list with all elements greater than the pivot 50.
- The pivot data 50, which is now at the right position in the array. This data can be ignored.

Quicksort is then applied to the left and the right sub-lists. The left sub-list used 25 as the pivot, and the right sub-list used 75 as the pivot. These two pivots are now at the right places. They are ignored.

(12 20 2 11 5 12) (**25**) (33 28) (**50**) (70) (**75**) (80 90 99 77)

The following shows the third recursive level. Value 12 was chosen as the pivot and both occurrences of 12 are gathered as the pivot.

(2 11 5) (**12 12**) (20) (**25**) (28) (33) ( ) (**50**) (70) (**75**) (77) (**80**) (90 99)

The following shows the fourth recursive level. Sub-lists with only one element are ignored.

( ) (**2**) (11 5) (**12 12**) (**20**) (**25**) (**28**) (**33**) (**50**) (**70**) (**75**) (**77**) (**80**) ( ) (**90**) (99)

The following is the fifth and the last recursive level.

(2) (5) (11) ( ) (12 12) (20) (25) (28) (33) (50) (70) (75) (77) (80) (90) (99)

Now all the sub-lists have at most one data object. The partition operations have been completed.

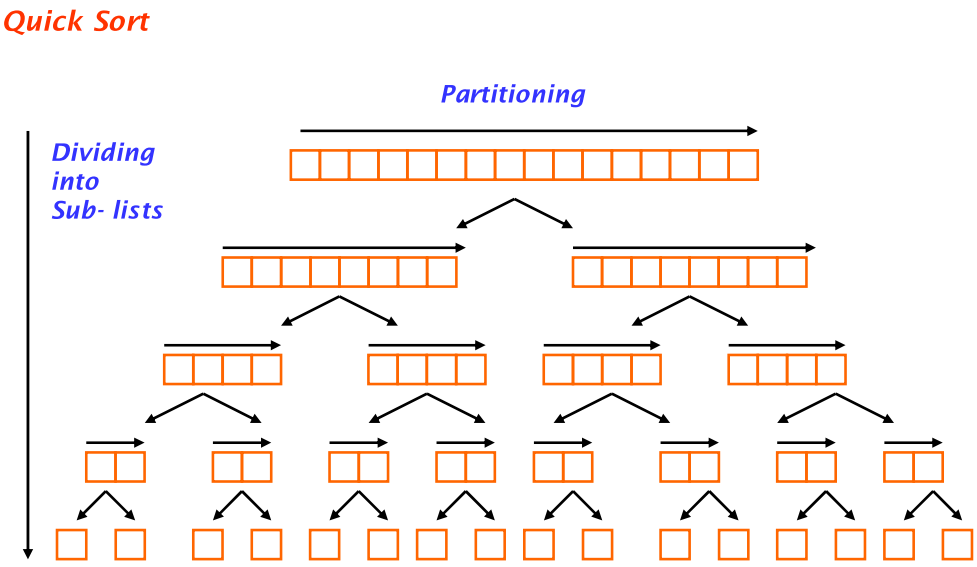
6.4.2 Performance Analysis of the Quicksort Algorithm

Quicksort can complete sorting the quickest if every list can be split into two equal sub-lists.

- The choice of pivot is very important.
- The best pivot is the one that divides a list into two equally long sub-lists.
- A middle value would be the best pivot.
- However, normally there is no way to know the middle value unless doing a traversal – which is time consuming and should be avoided.

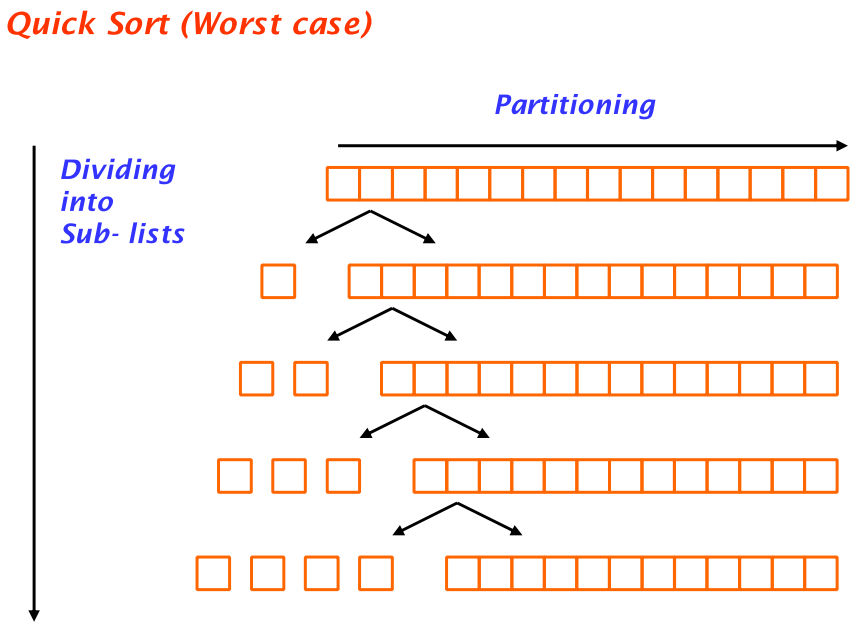
The normal approach is therefore picking the first element (or any element).

The following shows the **best-case partitioning** and the impact on the quicksort operation.



The following figure shows the **worst-case partitioning**, which is a very unlikely and unlucky situation.

- The pivot chosen is always almost the largest or smallest in the sub-list.
- Instead of creating two sub-lists of about equal size, one of the sub-list is almost as long as the parent sub-list.
- There are still a lot of partitioning and dividing operations to do before breaking down into sub-lists of size 1.



6.4.3 Implementation of the Quicksort Algorithm

Quicksort can be difficult to implement due to two reasons:

- It is recursively defined.
- It requires the creation of new lists in every step – potentially large memory consumption.

Example: quick_sort_1.py

```
# Quick sort (Ascending order)(New Lists)
def quickSortAsc(alist):
    size = len(alist)
    if size <= 1:
        return
    # create two new empty sub-lists
    smallList = list()
    largeList = list()

    # STEP 1: partitioning
    pivot = alist[0]
    pivotcount = 1   # potentially more than 1 pivot
    for i in range(1, size):
        if alist[i] < pivot:
            smallList.append(alist[i])
        elif alist[i] > pivot:
            largeList.append(alist[i])
        else:
            pivotcount += 1

    # STEP 2: recursive call to quicksort
    # the two sub-lists
    quickSortAsc(smallList)
    quickSortAsc(largeList)

    # now smallList and largeList are both sorted
    # STEP 3: copy the results to the original list
    alist.clear()
    alist.extend(smallList)
    alist.extend([pivot] * pivotcount)
    alist.extend(largeList)
```

Notes:

- This implementation creates two new lists for every partitioning operation.
- The first array element is always chosen as the pivot.
- The partitioning operation traverses the original array. It moves all data less than the pivot to `smallList` and moves all data greater than pivot to `largeList`.

6.4.4 (Challenging) In-Place Implementation of the Quicksort Algorithm

A more commonly used implementation of quicksort is so-called in-place sort implementation.

The function to call to quicksort becomes a simple one that creates a temporary list for the whole quicksort operation.

Example: quick_sort_2.py

```
# Quick sort (Ascending order)(In-place)
def quickSortInplaceAsc(alist):

    # create a temp list of sufficient size
    templist = [0] * len(alist)

    # start quicksort in another function
    quickSortInplaceHandle(alist, 0, len(alist) - 1, templist)
```

The function `quickSortInplaceHandle` then performs the actual recursive quicksort.

```
def quickSortInplaceHandle(alist, left, right, templist):
    if right <= left:
        return

    # partitioning
    pivotIndex = quickSortPartition(alist, left, right, templist)

    # recursive calls
    quickSortInplaceHandle(alist, left, pivotIndex - 1, templist)
    quickSortInplaceHandle(alist, pivotIndex + 1, right, templist)
```

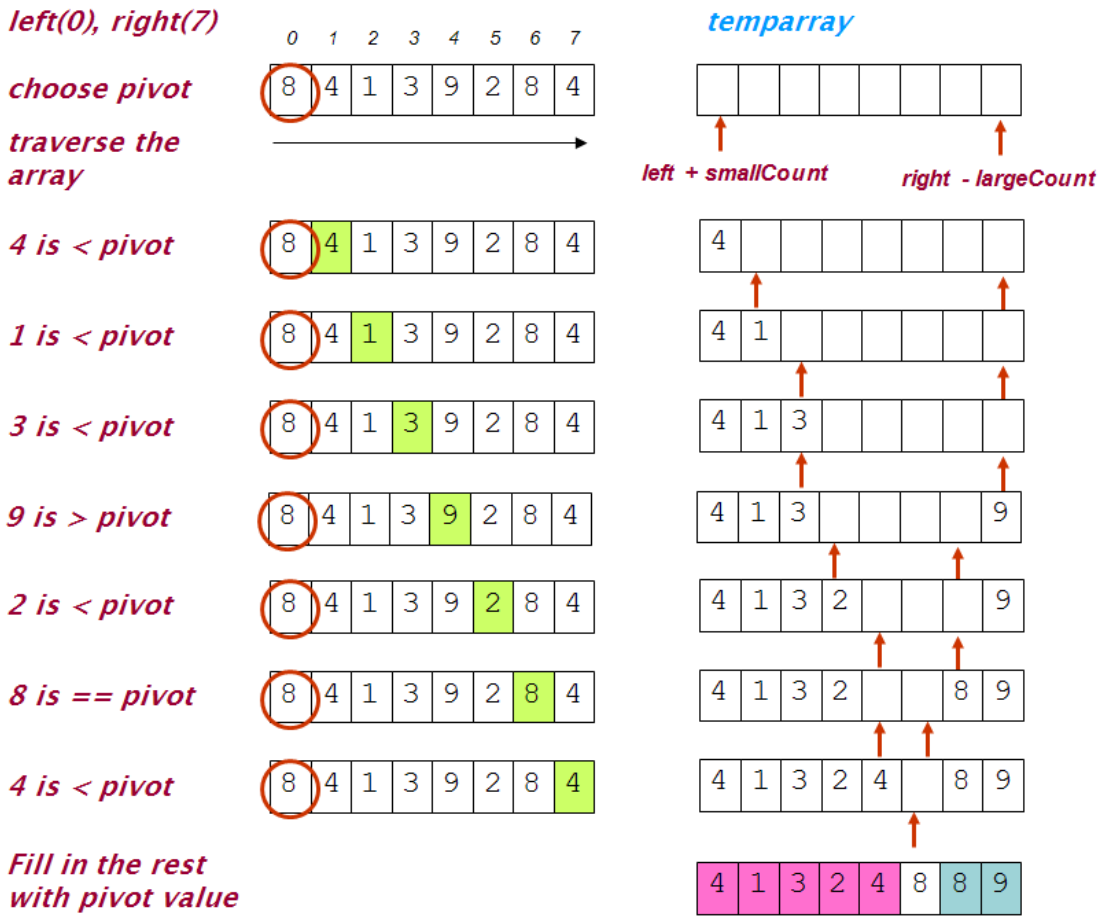
The function `quickSortPartition` is a helper function doing the partition.

```
# perform in-place partitioning of alist with the help
# of templist
def quickSortPartition(alist, left, right, templist):
    # parameters
    smallCount = 0   # number of data smaller than pivot
    largeCount = 0   # number of data larger than pivot
    pivot = alist[left]
    pivotcount = 1
    # partitioning
    for i in range(left + 1, right+1):
        if alist[i] < pivot:
            templist[left + smallCount] = alist[i]
            smallCount += 1
        elif alist[i] > pivot:
            templist[right - largeCount] = alist[i]
            largeCount += 1
        else:
            pivotcount += 1
    # add back the pivot
    # for loop needed because pivot may be more than 1
    for i in range(left + smallCount, right - largeCount + 1):
        templist[i] = pivot
    # copy back to original list
    for i in range(left, right + 1):
        alist[i] = templist[i]
    return left + smallCount
```

Notes:

- This implementation divides the solution into two functions: `quickSortInplaceHandle` and `quickSortPartition`.
- Partitioning now uses a temporary list for intermediate storage. This list is only created once in the function `quickSortInplaceAsc`
- This implementation is slower than the new list approach but using much less memory.
- This method works with array containing duplicated values (like 4 and 8 in the example below).
- A list copy operation is needed for moving data from `templist` back to the original `alist`.

The following diagram illustrates the operation. Note that the temparray is the templist.



6.4.5 Performance Analysis of the Quicksort Algorithm: Empirical Approach

The performance of quicksort is compared to the other sorting algorithms using the same experimental design.

Example: sort_analysis_2.py

```
from selection_sort_1 import sortSelectionAsc
from insertion_sort_1 import sortInsertionAsc
from bubble_sort_1 import sortBubbleAsc
from quick_sort_1 import quickSortAsc
import random
import time
import matplotlib.pyplot as plt

def genlist(size):
    return random.sample(range(0, size * 10), size)
```

```
if __name__ == "__main__":

    # experimental parameters
    dataSizeList = [1000, 2000, 3000, 4000, 5000, 8000, 10000, 16000]
    epoch = 10
    # experimental results
    SelSortResults = list()
    QuickSortResults = list()

    random.seed(1)
    for dataSize in dataSizeList:
        # TEST SELECTION SORT
        timeTaken = 0  # the total time for all epoches
        for e in range(0, epoch):
            # generate dataset
            numlist = genlist(dataSize)
            startTime = time.time()
            sortSelectionAsc(numlist)
            endTime = time.time()
            timeTaken += (endTime - startTime)
        timeTaken /= epoch
        print("[SELECTION] Time taken is {} s for data size {}".format(timeTaken, dataSize))
        SelSortResults.append(timeTaken)

        # TEST QUICK SORT
        timeTaken = 0  # the total time for all epoches
        for e in range(0, epoch):
            # generate dataset
            numlist = genlist(dataSize)
            startTime = time.time()
            quickSortAsc(numlist)
            endTime = time.time()
            timeTaken += (endTime - startTime)
        timeTaken /= epoch
        print("[QUICKSORT] Time taken is {} s for data size {}".format(timeTaken, dataSize))
        QuickSortResults.append(timeTaken)

    plt.xlabel('Data Size')
    plt.ylabel('Time Lapse (s)')
    plt.axis([0, dataSizeList[-1], 0, timeTakenResults[-1]])
    plt.plot(dataSizeList, SelSortResults, color='r', linewidth=2.0)
    plt.plot(dataSizeList, QuickSortResults, color='b', linewidth=2.0)
    plt.show()
```

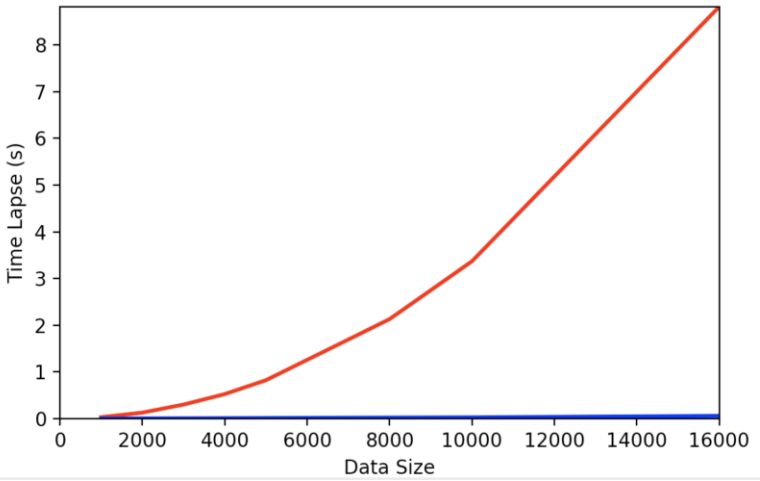
The following table compares the scalability performance of selection sort and quicksort. Remind that the scalability characteristic is more important than the absolute timing.

Sorting Algorithm	***1000 elements (s)***	***2000 elements (s)***	***4000 elements (s)***	***8000 elements (s)***	***16000 elements (s)***
Selection Sort	0.0303	0.124	0.515	2.142	8.778
Quick Sort	0.00205	0.00458	0.00975	0.0216	0.0460

Notes:

- Quicksort not only performs significantly better in absolute term.
- Quicksort also scales up much better than other sorting algorithms.
- For selection sort, doubling the task size will cause an approximately four times increase in time.
- For quick sort, the increase is a lot more graceful.

The following figure compares the scalability performance of selection sort (red) with quicksort (blue). Selection sort's performance shows a quadratic relation with the data size. However, the quicksort's performance does not seem to be affected much by the data size.



6.4.6 Performance Analysis of the Quicksort Algorithm: Analytical Approach

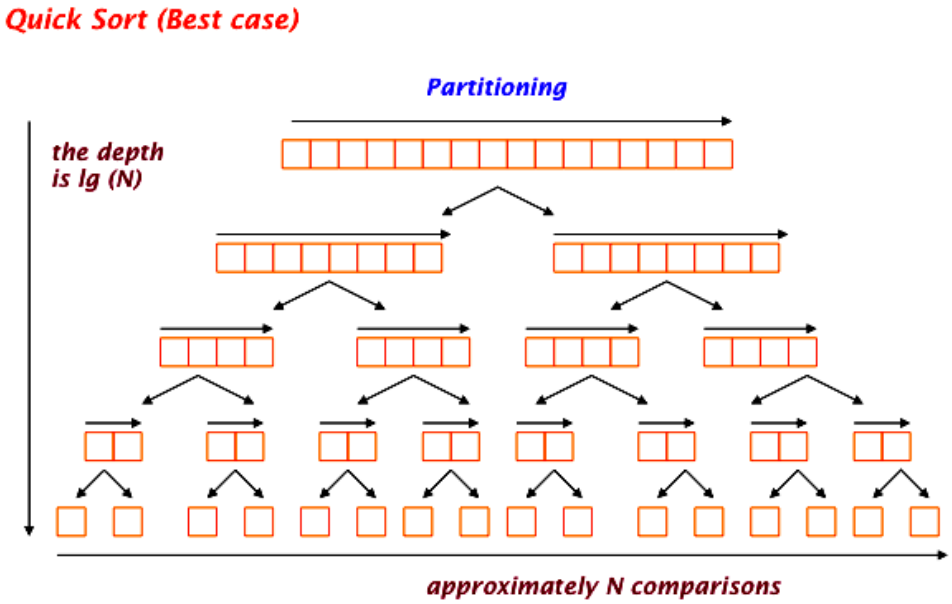
The empirical analysis seems to indicate that quicksort is significantly better than selection sort in scalability. This section discusses the performance of quicksort from the analytical approach.

The quicksort algorithm includes two operations: partition and recursive call. The performance of quicksort with respect to N (the number of data) can be estimated with counting the number of comparisons.

The partition step is based on an array traversal and comparison with the pivot. There are approximately N comparisons. The total number of comparison is worked out: multiplying N with the depth of the recursive calls.

The Best-Case Analysis

The following shows the best case: the level of recursive calls is minimized if the partitioning always split a list into two equal halves.

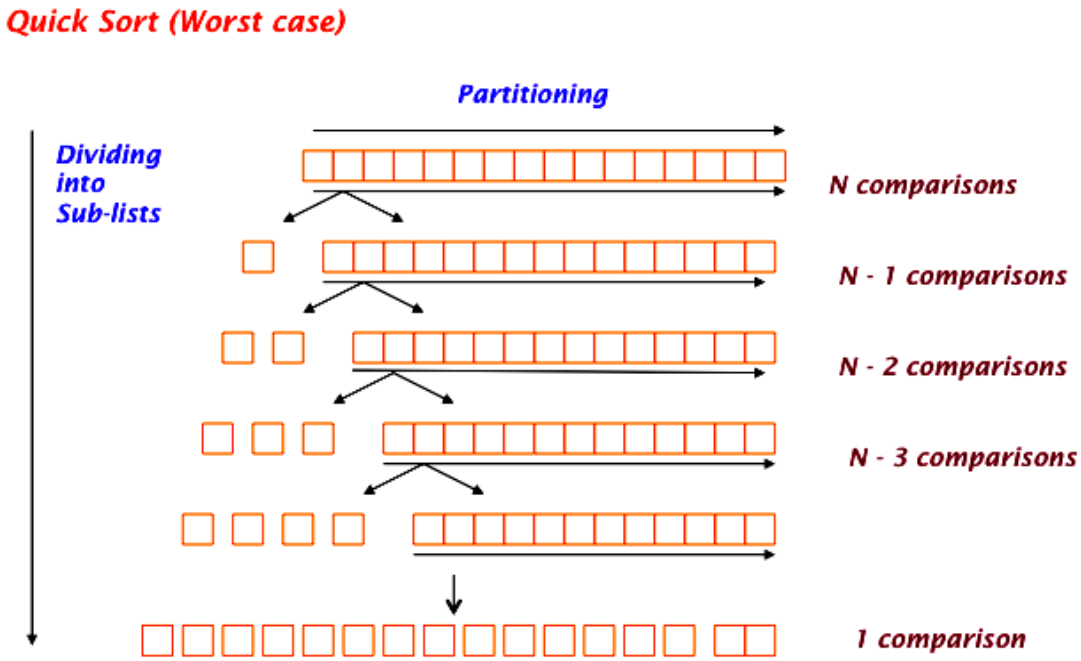


The depth of recursive call in the best case is lg (N). The derivation is the same as the case of binary search.

Total Number of Comparisons ≈ N lg(N) (when N is large)

The Worst-Case Analysis

The worst case happens when the partition step always produces one long sub-list and another empty sub-list.



The total number of comparisons is the sum of comparison in each partitioning

Number of Comparisons = 1 + 2 + ... + (N – 1) = N (N – 1) / 2

Number of Comparisons = 0.5 N^2 – 0.5 N

When N is large, the fast increasing N^2 term will make the N term insignificant.

Number of Comparisons ≈ 0.5 N^2 (when N is large)

The average case is approximately *N lg (N)*. No derivation is provided here.

Summary of Quicksort Performance Analysis

The following summarizes the performance of quicksort

Sorting Algorithm	*Best-case*	*Average-case*	*Worst-case*
Selection Sort	N^2	N^2	N^2
Quick Sort	*N lg (N)*	*N lg (N)*	N^2

COMPS209F Data Structures, Algorithms and Problem Solving

Basic Algorithm Analysis 7

The analysis of algorithms is to investigate the amount of time required to complete a task with the algorithms. The time is not the only factor of investigation. The amount of memory required is often the subject of investigation.

This chapter briefly discusses the topic of analysis of algorithms and gives a basic introduction of the important principles. If you will continue study computer algorithms, you will have opportunities to learn more.

7.1 Principles of Algorithm Analysis

Knowing the amount of time and memory required by an algorithm is important for the evaluation and comparison of performance of algorithms.

The usual approach of such investigation is to check how an algorithm can scale up to large amount of data. For example, it is common to investigate the time and memory requirement of an algorithm given data sizes of N, and then 2N, 4N, 8N, 16N, and so on. The scalability of the algorithm can be worked out from the change in performance in these different data sizes.

The investigation of the performance scalability is better than the investigation of absolute performance (or called benchmark testing). The absolute performance of an algorithm depends on other factors.

- Hardware power. An implementation on a supercomputer is clearly quicker than an implementation on a PC.
- Programming language. For example, the statement c = c + 1 is very common in many programming languages. Its C language equivalent statement c++, which may not work in other programming languages, executes a little bit faster than c = c + 1 in some old machines such as Digital's PDP11.

The following table illustrates that the first three sorting algorithms are essentially of the same growth rate. Double the data size increases the time taken four times. Quicksort clearly has a much slower growth rate.

Sorting Algorithm	***5000 elements (s)***	***10000 elements (s)***	***20000 elements (s)***	***40000 elements (s)***	***80000 elements (s)***
Selection Sort	0.0033	0.0131	0.0532	0.2099	0.8420
Insertion Sort	0.0030	0.0119	0.0476	0.1902	0.7652
Bubble Sort	0.0081	0.0329	0.1307	0.5146	2.0806
Quick Sort	0.00007	0.00017	0.00035	0.00073	0.00178

The following table shows the same implementation of the sorting algorithms running on a slower computer.

Sorting Algorithm	***5000 elements (s)***	***10000 elements (s)***	***20000 elements (s)***	***40000 elements (s)***	***80000 elements (s)***
Selection Sort	0.1000	0.4950	1.6510	6.5940	42.4680
Insertion Sort	0.0550	0.2350	0.9400	4.0360	30.2760
Bubble Sort	0.1830	0.6740	2.8030	12.0290	73.5650

The power of computers should not be a factor in performance analysis. On a slower computer, the absolute performance was worse, but the same growth rate is observed for all the algorithms.

The growth rate or the scalability analysis is a fairer metric for comparing algorithms.

7.2 The Big-O Notation

An algorithm is an operation mapping one set of input data to a set of output data. The time efficiency of an algorithm is how the time is related to the input data size N. Logically, more operations would be required for a larger input size.

7.2.1 Number of Operations as Estimation of Efficiency

The analytic approach of algorithm performance analysis uses the number of operations as an estimation of the time taken.

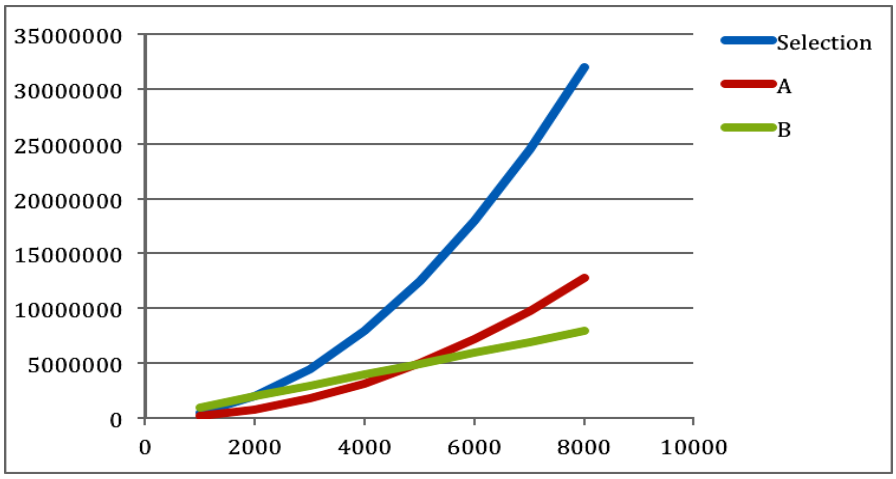
Algorithm	*Worst-case Operation Count (approx.)*
Sequential Search	$(1 + N) / 2$
Selection Sort	$0.5\ N^2 - 0.5\ N$
Algorithm A	$0.2\ N^2 + 100$
Algorithm B	$1000\ N + 5$

Consider that there are two other sorting algorithms, A and B.

- The number of comparisons of these two algorithms with respect to N is given in the above table.
- Which algorithm has comparable performance as the Selection Sort?
- Is there a better algorithm?

The absolute performance is irrelevant if N is not known. The focus should be the growth rate or the **scalability** of the algorithms.

The following shows the growth rate graphically.



Notes:

- Algorithm B has the most graceful growth rate, though it has a slightly poorer performance for small data size N.
- Selection sort and Algorithm A look to have a similar growth rate. Algorithm A is consistently better than selection sort.
- Algorithm B looks to have a linear scalability.
- Algorithm A and Selection Sort have the same scalability, and they are both super-linear.

7.2.2 Classes of Scalability Characteristics

The study of the growth rate of algorithms is more convenient if the growth rates are classified into a number of classes.

- Algorithms of different classes have significant different growth rate.
- Algorithms of the same classes may differ in the absolute performance, but their growth rates are comparable.

The following lists the major classes of performance of algorithms.

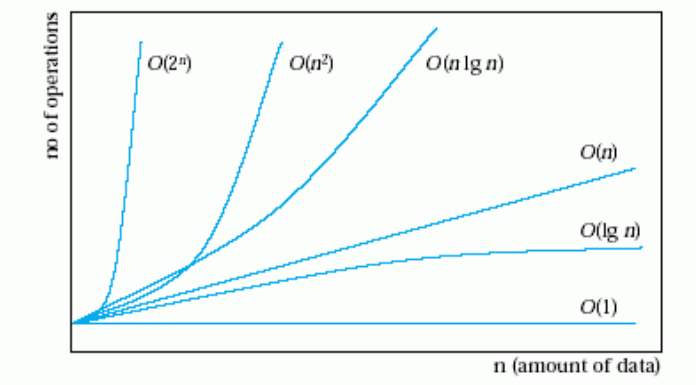
Classes	***Remarks***
Constant time, O(1)	The time is bounded by a constant and independent of N.
Linear time, O(N)	The time is directly proportional to N.
Quadratic time, $O(N^2)$:	The time is directly proportional to N^2.
Cubic time, $O(N^3)$:	The time is directly proportional to N^3.
Exponential time, $O(2^N)$:	The time is directly proportional to 2^N.
Logarithmic time, O(lg(N))	The time is directly proportional to lg N
Log-Linear time, O(N lg(N)):	The time is directly proportional to N lg N.

Notes:

- Constant time and Logarithmic time are sub-linear scalability.
- Log-Linear time is also known as Quasi-linear time. The performance can be considered as similar to Linear time.
- Quadratic time and Cubic time are known as Polynomial time. Polynomial time algorithms are considered to be feasible or tractable.

Generally, the slower the function grows, the less time is required for large amount of data. The growth of a function refers to the relative increment of the function between two values of N.

The following figure illustrates the growth rate of these functions.



Notes:

- The steeper the slope of the function is, the faster the function grows.
- The functions of N^3 and 2^N are growing more rapidly (slopes of the curves are getting steeper) compared to N^2 and N.
- However, the functions of N lg N and lg N are growing much slower than N^2.
- Logarithm base 2 is frequently used in computing environments rather than the common logarithm with base 10.
- Logarithm base 2 is written as lg instead of $\log_2$.
- Any logarithm (log) of N grows more slowly (as N increases) than any positive power of N.

Examples

Example: The built-in function max()

```
max(aList)
```

The built-in function max() finds the largest element in a list.

This algorithm is Linear time O(N) because the function traverses all elements of the list once.

Example: List Comprehension

```
newList = [random.randrange(0, dataSize) for n in
range(dataSize)]
```

This list comprehension runs a for loop N (i.e. variable dataSize) times

This algorithm is Linear time O(N).

Example: find_quadratic.py

```
def findQuadraticSolution(N):
    result = list()
    for x in range(1, N):
        for y in range(x, N):
            for z in range(y, N):
                if x**2 + y**2 == z**2:
                    result.append((x, y, z))
    return result
```

This function finds all solutions of the equation $x^2 + y^2 = z^2$

This algorithm is Cubic time $O(N^3)$ because there are three level of nested loop. The number of operations of the bolded line is approximately N^3

Example: Binary Search

```
def binarySearch(numlist, target):
    size = len(numlist)
    lower = 0
    upper = size - 1
    while lower <= upper:
        middle = (lower + upper) // 2
        if target == numlist[middle]:
            return middle
        elif target > numlist[middle]:
            lower = middle + 1
        else:
            upper = middle - 1
    return None
```

This function carries out binary search on a list for a target

This algorithm is Logarithmic time O(lg N). The size of the list is halved each iteration of the while loop.

7.2.3 Mathematical Definition of Big-O

The symbol O(N) is known as the big-O notation. This is used to put algorithms into different performance classes when the input size becomes larger.

For example, the worst case performance of sequential search is O(N) (pronounced as order N) and the worst case of selection sort is $O(N^2)$. For the algorithms A and B, their performance are expressed in big-O notation as $O(N^2)$ and O(N) respectively. So A and selection sort are of the same class.

Mathematically, an expression representing the time complexity with respect to N (say f(N)) can be expressed in big-O notation (say O(g(N))) with the following inequality.

$$f(N) < c\ g(N) \text{ for } N > N_0 \text{ and a constant } c$$

The big-O notation allows us to say that a function of N is *less than or equal to* another function g up to a constant as N grows to a very large number or infinity. By using the function f(N) = 100 N, we say that f(N) is O(N) because f(N) < c N for c = 100, g(N) = N and $N_0 = 1$. Any larger constant c > 100 and N > 1 will also work.

COMPS209F Data Structures, Algorithms and Problem Solving

Stacks, Queues and Linked-Lists 8

This and the previous Python course have covered several common used data structures or containers. They include Lists, Arrays, and Dictionaries. Each data structure has its characteristics.

Python List	Array	Dictionary
• Stores any data type • Heterogeneous (mixed types) • Indexed by position • Size is flexible and change is easy • Efficient access with position indexing	• Stores a specific data type • Homogeneous (same type) • Indexed by position • Size is fixed and change is expensive • Efficient access with position indexing	• Stores any data type • Heterogeneous (mixed types) • Indexed by an unique key of immutable type • Size is flexible and change is easy • Efficient access with key indexing

This chapter describes three important data structures for programming.

- Queues
- Stacks
- Linked Lists

They do not replace any of the above three data structures. In fact, Python's Lists and Dictionaries are very powerful that are adequate for almost all problems and programming needs. However, the same Python's Lists and Dictionaries are not available in all other languages.

These data structures are widely available in many languages. Knowledge in these data structures is important for expert programmers.

8.1 Queues and Stacks

Queues and stacks are both one-dimensional structure similar to Python's Lists. They are much more restrictive than Python's Lists in the aspects of data addition and retrieval.

- Queues: the data added and the data retrieve with a queue follows the rule of First In First out (FIFO).
    - The data that was added to a queue first will be the first one retrieved.
    - Just like the person joined a queue for a bus first will be the first one goes aboard the bus.
- Stacks: the data added and the data retrieve with a queue follows the rule of First In Last out (FILO).
    - The data that was added to a stack first will be the last one retrieved.
    - Just like a stack of plates on a table, that the first plate added (which should be at the bottom) will be the last plate removed.

Similar to other data models, stacks and queues define both data and operations. The names of the operations are defined.

Queues	Stacks
`enQueue`: adding data to the queue **`deQueue`**: retrieving data from the queue	**`push`**: adding data to the stack **`pop`**: retrieving data from the stack

8.1.1 Queues

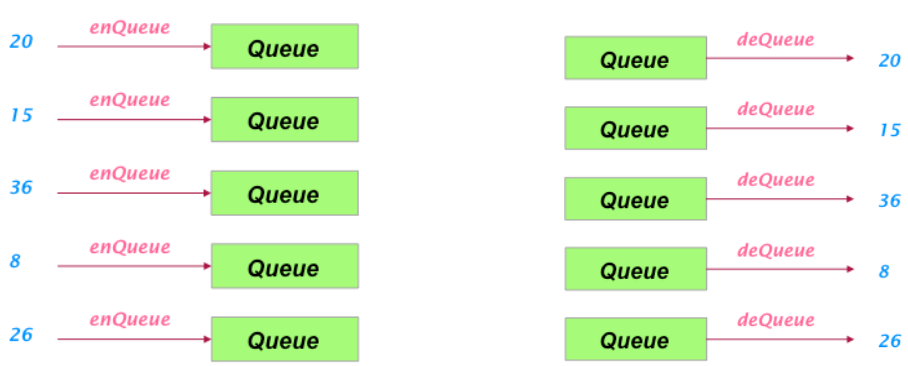
Queue is a data structure that is characterized by the order of data insertion and data retrieval. The first data added will be the first data retrieved. This is commonly called First-In-First-Out (FIFO).

There are two major operations for Queues.

- **`enQueue`**: adding data to the queue
- **`deQueue`**: retrieving data from the queue

The following example illustrates that the order of added data is the same as the order of retrieved data.

- The integers added through **`enQueue`**: 20, 15, 36, 8, 26
- The integers retrieved through **`deQueue`**: 20, 15, 36, 8, 26



Queues are found in many real life situations where the order of data insertion and retrieval is important.

- The order of arriving at a bus stop should determine the order of boarding the bus
- The order of making a booking to a facility should determine the priority of using the facility
- The order of requesting the single hard disk should determine the order of using the hard disk

Implementation of Queue with Python's List

Queues can be implemented in many ways. The following shows a Queue class that is implemented with a Python's list as the data model.

Example: queue.py

```
class Queue:
    def __init__(self):
        self.data = list()  # create an empty list

    def enQueue(self, value):
        self.data.append(value)

    def deQueue(self):
        if len(self.data) == 0:
            raise IndexError('Attempt to deQueue an empty queue')
        value = self.data.pop(0)
        return value

    def isEmpty(self):
        return len(self.data) == 0

    def isFull(self):
        return False

    def print(self, sep=','):
        isFirst = True
        for value in self.data:
            if not isFirst:
                print(sep, end='')
            print(value, end='')
            isFirst = False
        print() # add a new line at the end
```

Notes:

- The __init__ method has created a list.
    - The queue's data needs an actual data container for storage.
    - The list is used for storing actual queue data.
- The **`enQueue`** and **`deQueue`** methods use appropriate list methods to achieve First-In-Last-Out.
    - The **`enQueue`** method adds data to the end of list (using **`append`**).
    - The **`deQueue`** method retrieves and remove data from the front of the list (i.e. index 0).
- The **`deQueue`** method raises an exception if the list is empty.
- The `isEmpty`, `isFull` and `print` are auxiliary methods for checking the queue.

The following shows an example of how to use the Queue class.

Example: queue.py

```
if __name__ == "__main__":

    theQueue = Queue()
    theQueue.enQueue(20)
    theQueue.enQueue(15)
    theQueue.enQueue(36)
    theQueue.enQueue(8)
    theQueue.enQueue(26)
    print("Content of the queue: ", end='')
    theQueue.print()

    while not theQueue.isEmpty():
        print(theQueue.deQueue())
```

```
Content of the queue: 20,15,36,8,26
20
15
36
8
26
```

Notes:

- The order of the integers retrieved from the queue is same as the order of adding the integers.
- The method `isEmpty` is useful as the stopping condition for iteration.

8.1.2 Stacks

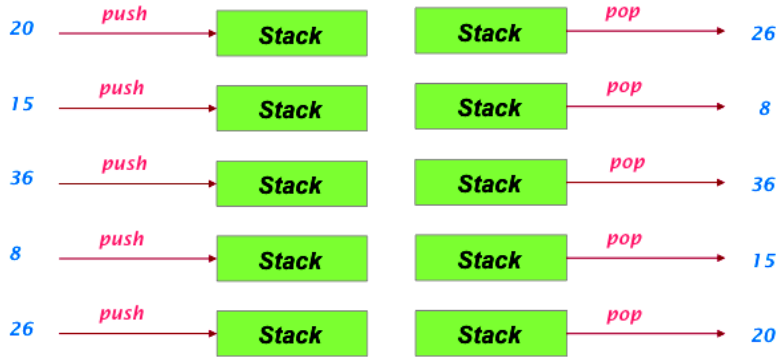
Stack is also a data structure that is characterized by the order of data insertion and data retrieval. The first data added will be the last data retrieved. This is commonly called First-In-Last-Out (FILO).

There are two major operations for Queues.

- **`push`**: adding data to the queue
- **`pop`**: retrieving data from the queue

The following example illustrates that the order of adding data is the reverse of the order of retrieving data.

- The integers added through **`push`**: 20, 15, 36, 8, 26
- The integers retrieved through **`pop`**: 26, 8, 36, 15, 20



Stacks are found more often in computing operations than in real life situations.

- The stacking of dishes is one example.
- Dishes stacked up are given in the reverse order the first dish is used last.

In computer operations and programming, stacks are quite common.

- For example, the runtime stack that supports local variables creation and function calls is one.
- Stacks are also found in recursive problems, string and text processing, and push down automata.

Implementation of Stack with Python's List

Stacks can be implemented in many ways. The following shows a Stack class that is implemented with a Python's list as the data model.

The implementation is very similar to that of the Queue class. There is only one difference:

- For queues, the data is added at the end of the list. The data is retrieved from the front of the list.
- For stacks, the data is added at the end of the list. The data is retrieved from the end of the list.

Example: stack.py

```
class Stack:
    def __init__(self):
        self.data = list()  # create an empty list

    def push(self, value):
        self.data.append(value)

    def pop(self):
        if len(self.data) == 0:
            raise IndexError('Attempt to pop an empty stack')
        value = self.data.pop()  # from last index
        return value
...
```

Notes:

- The only significant change from the Queue class is the bold statement.
    - The Stack class removes data from the last index.
    - The Queue class removes data from the first index.

The following shows an example of how to use the Stack class.

Example: stack.py

```
if __name__ == "__main__":

    theStack = Stack()
    theStack.push(20)
    theStack.push(15)
    theStack.push(36)
    theStack.push(8)
    theStack.push(26)
    print("Content of the stack (top at the end): ", end='')
    theStack.print()

    while not theStack.isEmpty():
        print(theStack.pop())
```

```
Content of the stack (top at the end): 20,15,36,8,26
26
8
36
15
20
```

Notes:

- The order of the integers retrieved from the queue is the reverse of the order of adding the integers.

8.2 Linked Lists

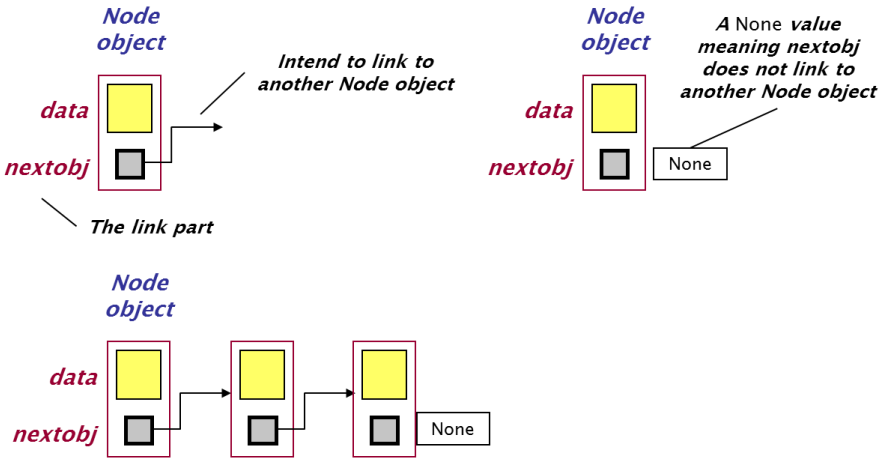
Linked lists are very flexible in their size.

- Linked lists are built from **nodes**.
- One node can store one value.
- To add a value to the linked list, add one node to the linked list for the new value.
- To remove a value from the linked list, remove the corresponding node from the linked list.

8.2.1 Nodes of Linked Lists: Self-Referencing Structure

Nodes of linked lists are a **self-referencing data structure**.

- Each node consists of a **data part** (i.e. payload) and the **link part**.
- The link part can store a reference to another node.



The above diagram illustrates what a self-referencing structure can do.

- A node contains a data part and the link part (top).
- The link part can contain reference to another node (bottom).
    - One node can link to another node, which can link to the third node.
    - A number of nodes are therefore linked together, which is called a linked list.
- The linked nodes should have an end.
    - The end is signified when the link part of a node is assigned a special value, such as `None`.
    - The node with `None` is called the end node or the last node.

Class Definition in Python

The following shows a Python class definition for nodes for linked lists.

Example: linked_list_1.py

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.nextobj = None
```

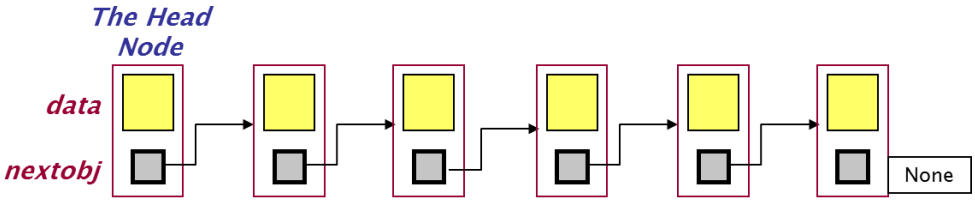
Notes:

- The __init__ method creates two class properties.
    - The property **`data`** will hold data (i.e. payload).
    - The property **`nextobj`** will hold reference of the next node. It is defaulted to **`None`**, meaning that no node follows.

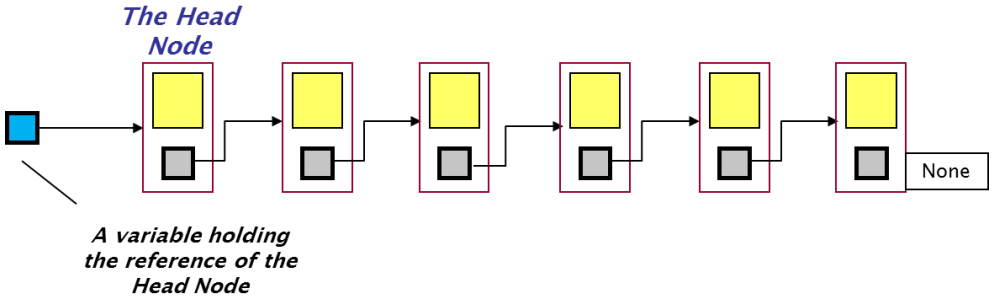
8.2.2 Managing Linked Lists: Head Node, Node Addition, and List Traversal

The first node, which is often called a **head node**, of a linked list is most important.

- The nodes are objects.
- Every object in Python should be assigned to a variable so that the object can be used.
    - The variable holds the reference of the object.
- In a linked list, every node's reference is held by the pervious node (see the figure below).
    - Except the head node. The head node has no previous node.



- Only through the head node, all other nodes of the link list can be used.
    - The process starts from the head node, and then the next node, and so on.
    - Each node is visited one after another.
    - This process is called **traversal**.
- Every linked list needs an additional variable for holding the reference of the head node (see figure below).



Example: Building a Simple Linked List

Any effective data structure should provide methods to add data, retrieve data, and remove data, and so on. Adding new data to a linked list involves adding a new node.

The following example shows how to add three nodes to a linked list.

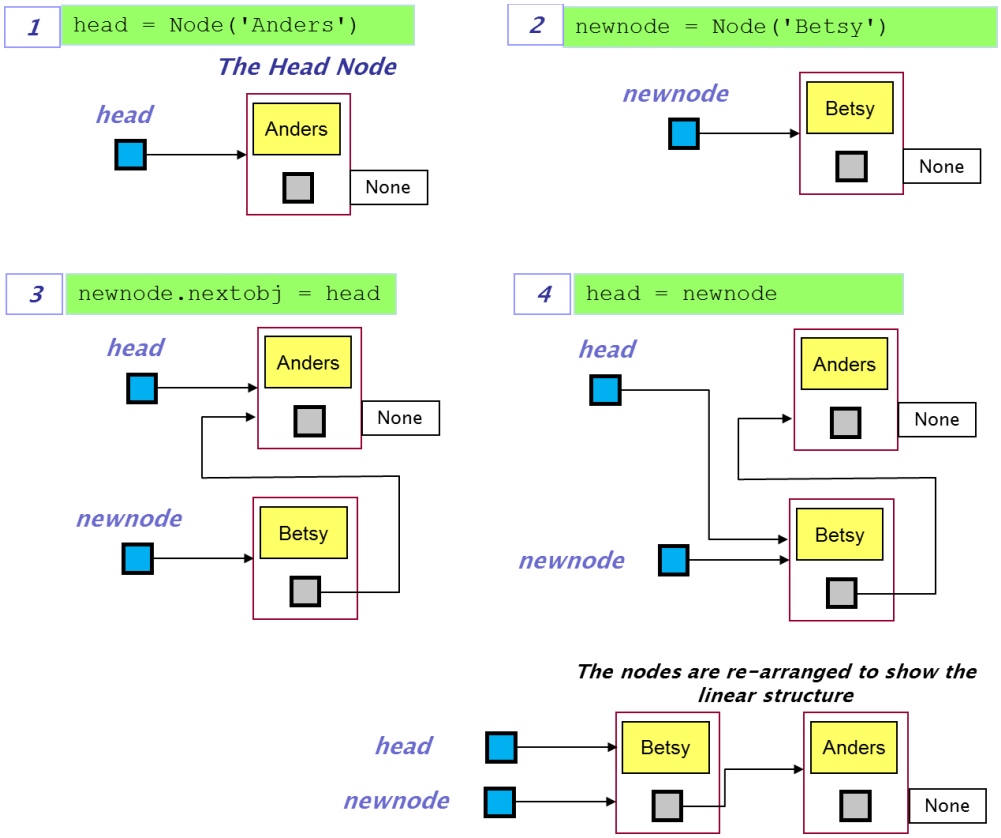
Example: linked_list_1.py

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.nextobj = None

# create the first node
# use head to hold the reference of the node
head = Node('Anders')

# create the second node
newnode = Node('Betsy')
newnode.nextobj = head
head = newnode

# create the third node
newnode = Node('Chris')
newnode.nextobj = head
head = newnode
```

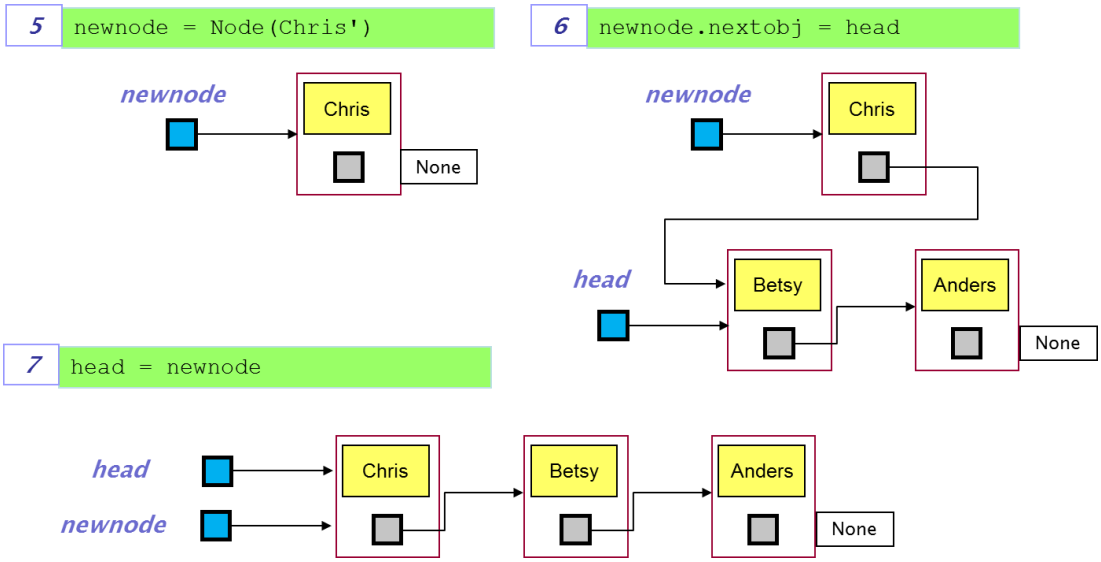
Notes:

- The method is to build the first node and assign its reference to the variable **`head`**.
- The next two new nodes are inserted in the front.
    - Create new node with initialization of a name.
    - Attach the existing linked list (i.e. the head of) to the new node's `nextobj`.
    - The new node becomes the head node, and therefore updates the variable **`head`**.

The following figure shows the details of creating the first two nodes.



The creation of the third node follows the same pattern.



Notes:

- The newest node added is inserted at the front.
- The first (oldest) node added will be found at the end of the linked list.

Example: Traversing a Linked List

Unlike arrays, direct access to a particular linked list node is not allowed. The exception is the head node. To access a node, a process called traversal is needed.

A linked list traversal first requires the reference to the head node. Then move from the head node to the second node, and then to the third node and so on.

Traversal is based on a simple while structure. An implementation is given below.

Example: linked_list_1.py

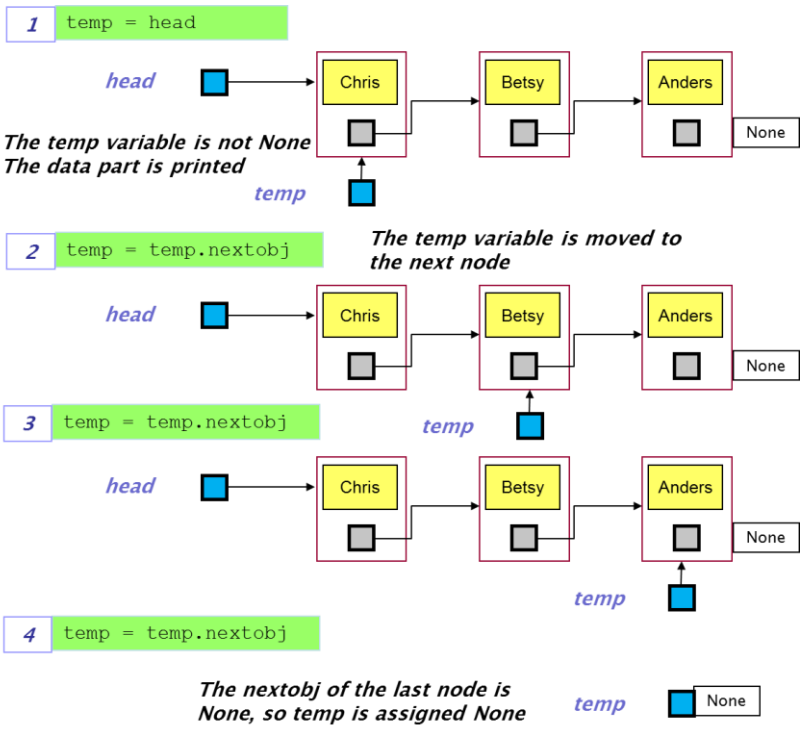
```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.nextobj = None
...

# traverse the nodes
temp = head
while temp != None:
    print(temp.data)
    temp = temp.nextobj
```

```
Chris
Betsy
Anders
```

Notes:

- Assume that the variable **`head`** is holding the reference to the head node.
- The reference to the head node is copied to a variable `temp`.
  - This avoids changing the variable `head`. The reference to the head node is lost if it is changed.
  - The `temp` variable is *travelling* from the first node to the last node. It means that the `temp` variable is holding the reference of each node from the first to the last.
  - At each node, the data part of the node is printed.

The following figure illustrates the process of traversal and the work of the variable `temp`.



8.2.3 Implementation of a Linked List ADT

This section describes a linked list ADT based on the self-referencing node structure described before.

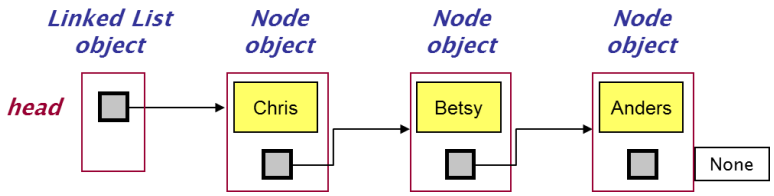
Basic Implementation

A basic implementation defines two classes: `Node`, which is described previous, and `LinkedList`, which is new.

The `LinkedList` class has the following purposes:

- Holds the reference to the head node of a linked list.
    - It is also called a placeholder.
- Offers methods for linked list computation

An actual linked list consists of one object of `LinkedList` and many objects of `Node`.



The basic implementation is shown below.

Example: linked_list_adt.py

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.nextobj = None

class LinkedList:
    def __init__(self):
        self.head = None   # the head variable is set to None

    def insert(self, data):  # insert new data at the front of linked list
        newnode = Node(data)
        newnode.nextobj = self.head
        self.head = newnode

    def print(self):   # printing all nodes by traversal
        temp = self.head
        while temp != None:
            print(temp.data)
            temp = temp.nextobj
```

The following illustrates how to use the Linked List ADT.

Example: linked_list_adt.py

```
...

linkedlist = LinkedList()
linkedlist.insert('Anders')
linkedlist.insert('Betsy')
linkedlist.insert('Chris')
linkedlist.print()
```

```
Chris
Betsy
Anders
```

Adding Node at the End of Linked List

Adding data to the end of linked list needs more work.

- Traverse the nodes until the last node is reached.
- Create a new node.
- Attach the new node to the last node.

The following method adds a new node at the end of linked list.

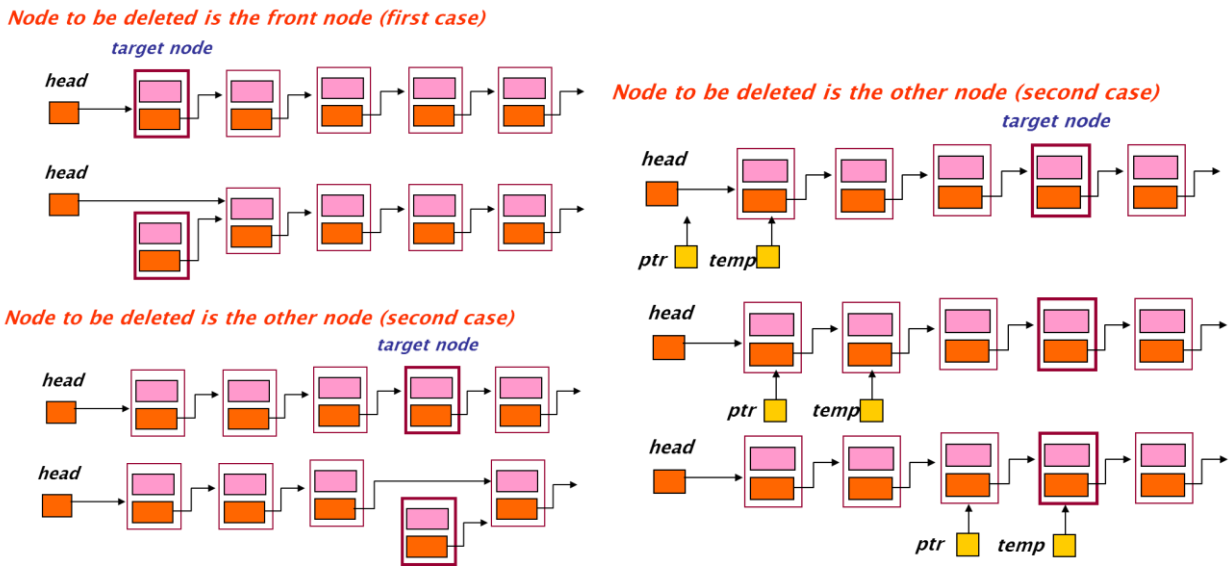
Example: linked_list_adt.py

```
...
    def insertEnd(self, data):  # insert new data at the end
        if self.head is None:       # if linked list has no node
            self.head = Node(data)  # simply add a new node
            return
        temp = self.head        # traverse to the last node
        while temp.nextobj != None:
            temp = temp.nextobj
        # temp is now at last node
        temp.nextobj = Node(data)
```

Deleting Nodes in Linked List: Based on Position

Deleting data from linked list needs a sequence of careful operations.

- There are two cases to consider (see left in below figure)
    - If the target node to delete is the front node, the head variable must change.
    - If otherwise, the head variable is not affected.



- Search for the node to be deleted (based on position)
    - Searching is based on traversal of nodes using two variables `temp` and `ptr`.
    - The variable `temp` hopes to traverse to the target node.
    - The variable `ptr` is always the previous node of `temp`.

Example: linked_list_adt.py

```
...
    def delete(self, index):   # index is the position of the target node
        if self.head is None:
            return
        if index < 0:
            raise IndexError("Attempt to use negative index")
        # first case
        temp = self.head
        if index == 0:
            self.head = temp.nextobj
            del temp
            return
        # second case
        while index > 0:
            ptr = temp
            temp = temp.nextobj
            index -= 1
        ptr.nextobj = temp.nextobj
        del temp
```

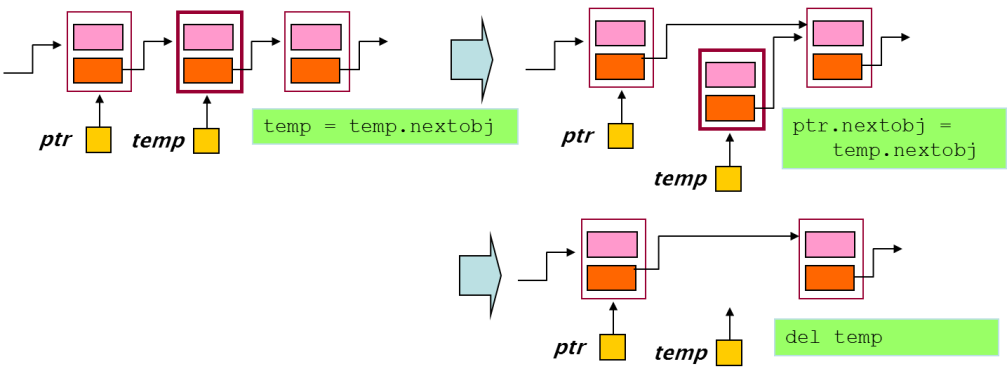
Notes:

- The removed nodes must be destroyed with the **`del`** keyword.

  ```
  del temp
  ```

- The second case involves two steps.
  - Traverse the nodes until the target node is reached.
  - Remove the node from the linked list and destroy it.

Details of operations (second case)



Deleting Nodes in Linked List: Based on Data

The implementation is similar, except the target node is looked up with data comparison.

Example: linked_list_adt.py

```
...
    def deleteNodeOfData(self, targetdata):
        if self.head is None:
            return
        # first case
        temp = self.head
        if temp.data == targetdata:
            self.head = temp.nextobj
            del temp
            return
        # second case
        while temp != None and temp.data != targetdata:
            ptr = temp
            temp = temp.nextobj
        if temp is None:
            return
        ptr.nextobj = temp.nextobj
        del temp
```

8.2.4 Linked Lists: Performance Analysis

Common operations of linked lists include data addition, searching, and data removal. The time complexity classes of major operations of linked lists are given in the following table.

Operations	Average-case	Worst-case
Node addition (at front)	O(1)	O(1)
Node addition (at back)	O(N)	O(N)
Searching	O(N)	O(N)
Traversal	O(N)	O(N)
Access to a Specific Node	O(N)	O(N)
Node deletion	O(N)	O(N)

Notes:

- Adding a new node at the front does not require traversal.
    - Constant time operation (not dependent on the amount of data).
- Compared to arrays, linked list is worse in accessing to a specific node.
    - Arrays support the indexing operation, which provides a direct access to an element given the index.
    - Linked lists require a traversal operation in order to access a certain node.
- Data searching is also not as good as arrays.
    - Sorted arrays support very efficient data searching with the binary search algorithm.

COMPS209F Data Structures, Algorithms and Problem Solving

Binary Trees 9

Binary trees are similar to linked lists.

- The size is flexible.
- The basic building block is a self-referencing node.

However, binary trees provide additional features compared to arrays and linked list.

- Binary trees are non-linear structures.
- The nodes in binary trees are arranged in a tree-like manner.
- Each node is linked to two other nodes in a parent-child relation.
- The two children nodes also take up specific positions of left and right.
- This provides a means to arrange data to facilitate efficient searching and sorting algorithms.

9.1 Introduction to Binary Trees

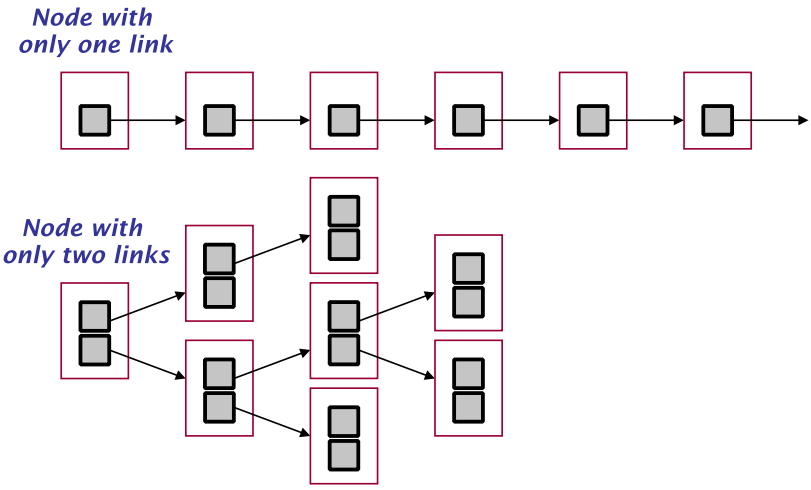
Binary trees are non-linear structures.

- Data organized in a non-linear structure are not in one-dimensional manner.
- Data are arranged in a tree-like structure in binary trees.
- Very efficient at data searching.

Both binary trees and linked lists are built with self-referencing node.

- Nodes for linked lists have one link property, which links to one node.
- Nodes for binary trees have two link properties, which links to two nodes.

Nodes with one link property can only be built into a linear structure. However, nodes with two link properties can be built into non-linear structures. The following figure illustrates the difference.



Class Definition in Python

The following shows a Python class definition for nodes for binary trees.

Example: binary_tree_1.py

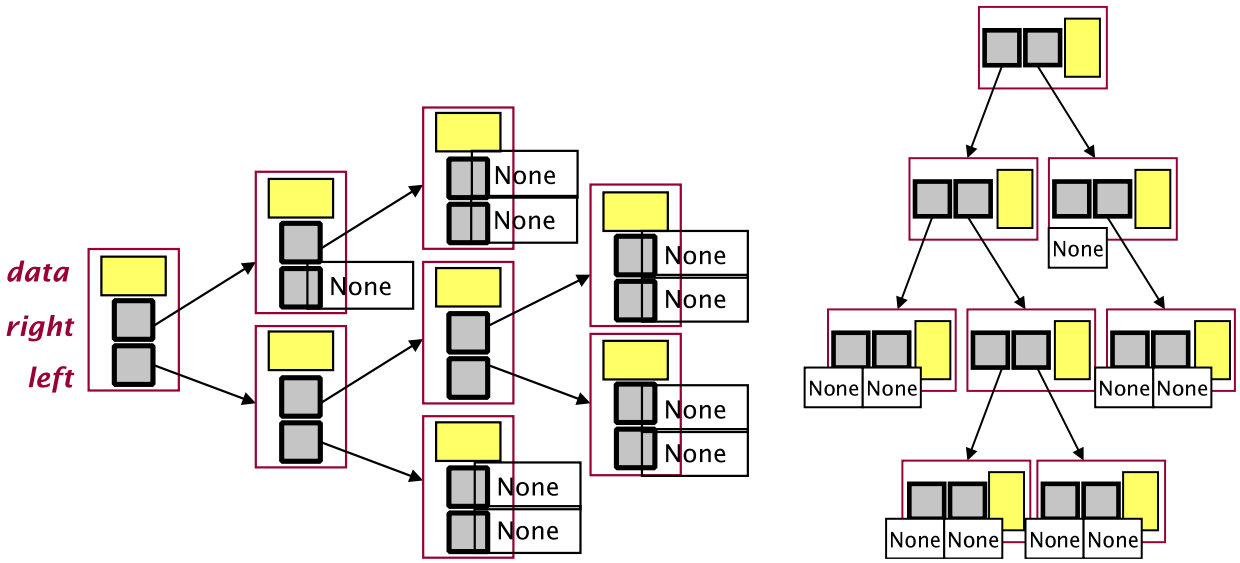
```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.left = None
        self.right = None
```

Notes:

- The `__init__` method creates three class properties.
  - The property **`data`** will hold data (i.e. payload).
  - The properties **`left`** and **`right`** will hold reference of another node. They are defaulted to **`None`**, meaning that no node is linked.

The nodes based on this Python class definition can connect to at most 2 nodes.

- The 2 nodes are the maximum.
- A node may connect to no node, or connect to one node on the left, or connect to one node on the right.

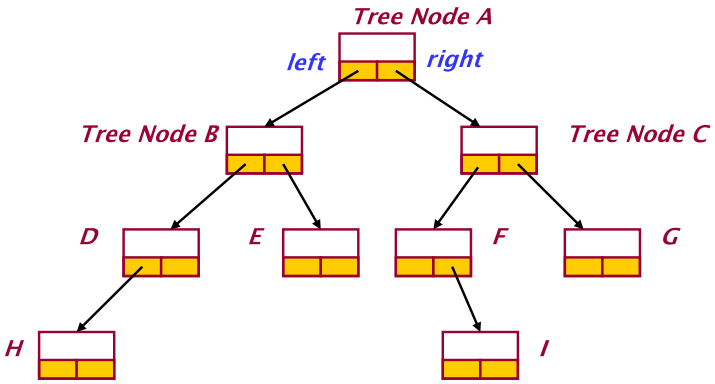


These are binary trees. The one on the left looks like a tree lying sideway. However, normally trees are drawn upside down (right in the above figure).

Relations within a Binary Tree

In a linear structure, there are only two relations between nodes: before and after. A node is before another node, or a node is after another node.

In a binary tree, there are many more relations between nodes. More relational information can therefore be encoded within a binary tree. Consider the following tree.



Notes:

- A **parent node** can connect to at most 2 **children nodes** through the left link and right link.
  - The 2 nodes are called **left child** and **right child** respectively through the left link and right link.
  - For example, node A is the parent node of node B and node C.
  - On the other hand, node B and C are left child and right child of node A respectively.
- A **root node** has no parent node.
- A **leaf node** has no child node.
- **Siblings** are two nodes with a common parent node.
- An **internal node** is a non-leaf node.

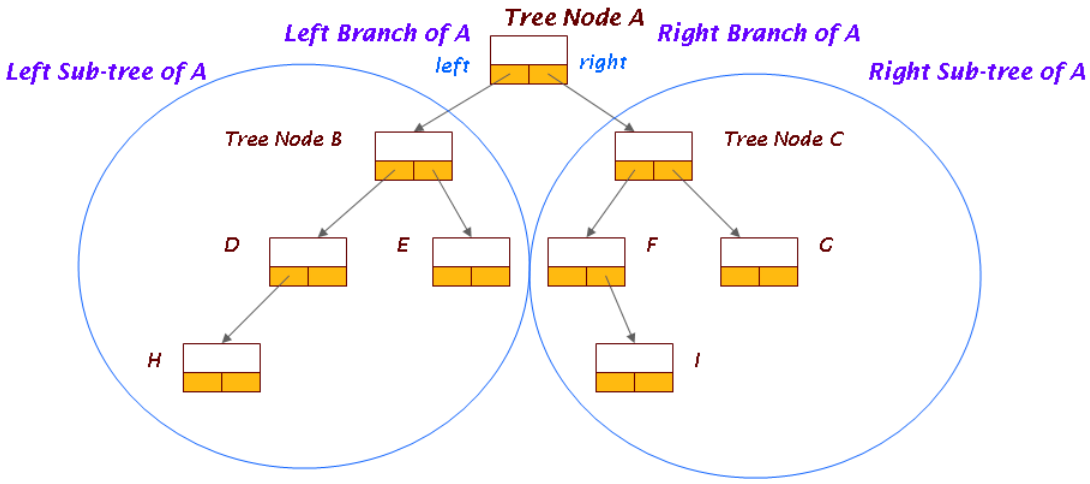
The following table lists the terminologies used regarding the node types of binary trees.

Node Type	***Remarks***	***Examples***	***Other Names***
Root Node	It has no parent node	Node A is a root node	
Leaf Node	It has no child node	Nodes H, E, I, and G	Terminal Node, External Node
Sibling Nodes	Nodes with a common parent	Nodes B and C are siblings	
Internal Node	It is not a leaf node	Nodes A, B, C, D, F	Inner Node

Binary Tree as a Recursive Structure

Binary trees have a recursive structures.

- A binary tree is composed of many smaller binary trees.
- For every node, the left and the right child are the root nodes of two binary trees.
- These two binary trees are called the **sub-trees**.

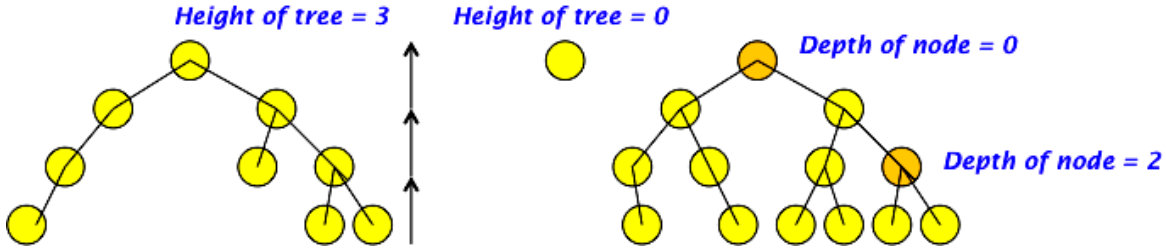


Notes:

- The left child node (i.e. node B) of node A is the root node of the left sub-tree of node A.
- The right child node (i.e. node C) of node A is the root node of the right sub-tree of node A.
- The left and the right sub-trees are also binary trees.
- The path connecting one node to another node is called an **edge**.
    - Each edge is a path of unit length.
    - The left edge is called the **left branch**.
    - The right edge is called the **right branch**.

Depth and Height of Binary Trees

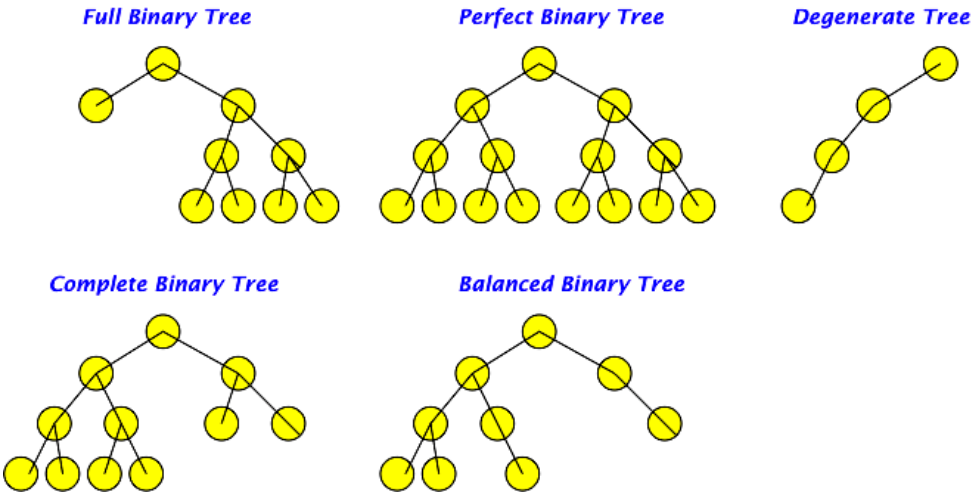
The **depth** of a node is the length of the path from the root node. The **height** of a tree is the path length from the deepest node to the root node.



Types of Binary Trees

The following table lists the terminologies used to describe different types of trees.

Binary Tree Types	***Remarks***
Full Binary Tree	All nodes, except the leaf nodes, have exactly two children
Perfect Binary Tree	A Full Binary Tree AND all leaf nodes are at the same level
Degenerate Tree	All nodes, except the leaf node, have exactly one child. It basically performs like a linked list
Complete Binary Tree	At all depth are completely filled with nodes, except for the deepest level
Balanced Binary Tree or AVL Tree	For each node, the height of the left sub-tree and the height of the right sub-tree differs by at most 1
Empty Tree	A binary tree has no node



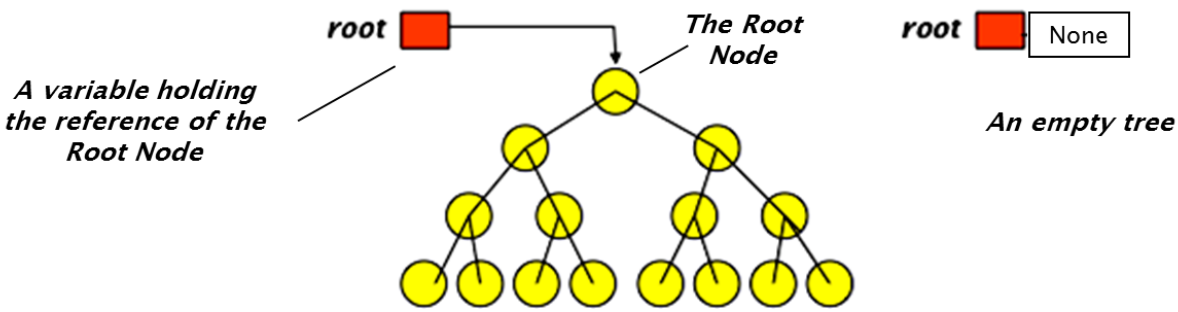
Most Important Node of Binary Trees

The **root node** is the most important node of binary trees.

- Only through the root node, all other nodes of the binary tree can be visited by traversal.
- The reference of the root node should be securely stored away in a variable.

The following figure shows the use of a variable to store the reference of the root node of a tree.

- A special value None is used to indicate that there is no node in the tree – an **empty tree**.



9.2 Binary Search Trees

There is one key difference between binary trees and **binary search trees (BST)**. In binary search trees, the nodes are arranged in order.

- It means that the nodes are sorted.
- The chapter on searching has informed that a data structure with an order facilitate efficient searching.

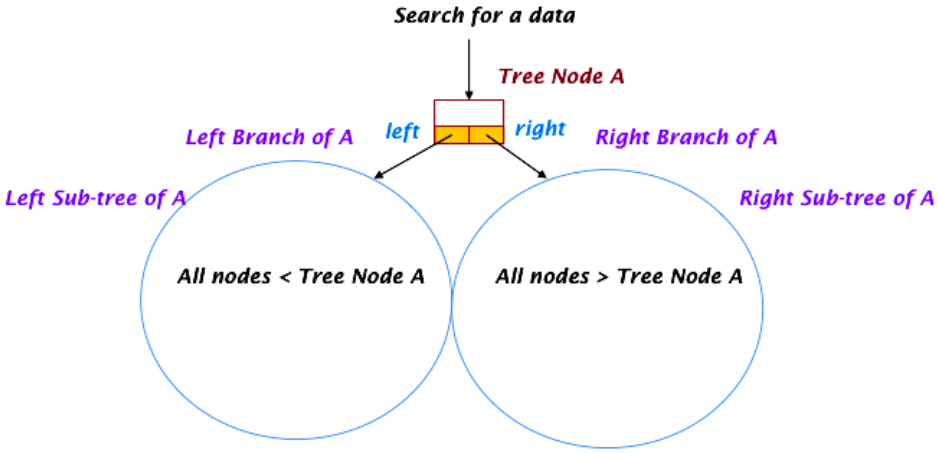
9.2.1 The Golden Rule of Binary Search Trees

The arrangement of the nodes in a binary search tree obeys the following rule:

children in left sub-tree < ***Node*** < *children in right sub-tree*

In a binary search tree (in ascending order), every node is always of a higher order than all children in the left branch and always a lower order than all children in the right branch.

This is called the golden rule of binary search tree.



Notes:

- The golden rule of binary search trees applies to every node.
- If every node follows the rule, then for every node, all nodes in the left sub-tree must be of a lower order than the node.
- All the nodes in the right sub-tree must be of higher order than the node. This property allows very efficient searching algorithms.

The Key of Nodes of Binary Search Trees

A **key** must be specified for the nodes of a binary search tree.

- The ordering of nodes is made according to the key of the nodes.
- The key can be part of the data or derivation from the data.

The structure of a node for a binary search tree should consist of three parts.

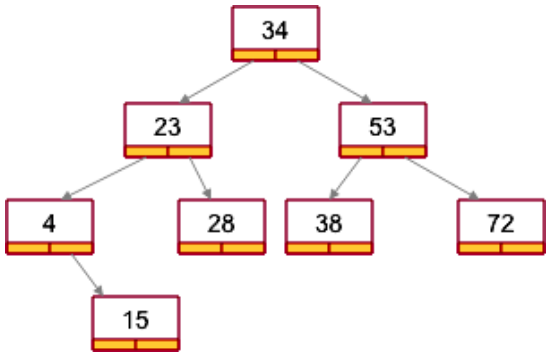
- The key: a particular field of the data can be designated as the key.
- The data: this part could be absent.
- Linkage: consists of the left link and the right link.

Data Searching in Binary Search Trees

BSTs are very efficient in data searching based on the key.

- The key is usually selected that can uniquely identify the data nodes.
    - For examples, HKID that can uniquely identify any person in HK, OUID for identification of students.
- The nodes are sorted according to the key.
- Each key provides a direction to find the location of the target key.

Consider the following binary search tree, of which only the key is shown on every node.



Notes:

- Searching for a target key involves traversal starting from the root node, which is 34.
- Traversal involves moving from one node to another node
    - At each node, there are two ways to go: left or right.
- For the node 34, all nodes to the left branch are smaller, and all nodes to the right branch are large.
- If the target key is X, and X is larger than 34, then the correct direction is the right link.
    - On the other hand, if X is smaller than 34, then the left link is the correct direction.

9.2.2 Searching in Binary Search Trees

There are two types of data search operations, depending on the target.

- Key searching: the target is a key or based on the key of nodes
- Non-key searching: the target is based on the data part.

Key Searching in BST

Binary search trees are very efficient in key searching.

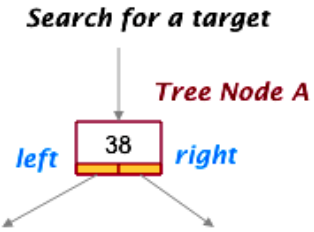
- The nodes in binary search trees are arranged in an orderly manner that could be exploited in data search.
- Each node has become a signpost:
    - (The key in the) children nodes in left sub-tree are all smaller
    - (The key in the) children nodes in right sub-tree are all greater

Data searching is always based on the process of traversal with the root node as the starting point. A matching is made at each node to see if the target is found.

- If the target is found, searching is successful.
- If the target is smaller, then the left sub-tree should be searched.
    - The right sub-tree can be ignored.
- If the target is larger, then the right sub-tree should be searched.
    - The left sub-tree can be ignored.
- Each comparison can effectively eliminate up to half the nodes from further searching operation.

This characteristic is very similar to binary search.

- Binary search replies on a sorted array in which every element is also a signpost.
- The smaller values are on the left and the larger values are on the right.



Algorithm of Key Searching in BST

Algorithms can be written in pseudo-code for easier understanding.

The following shows the algorithm of key searching in BST.

Algorithm: Key searching in BST

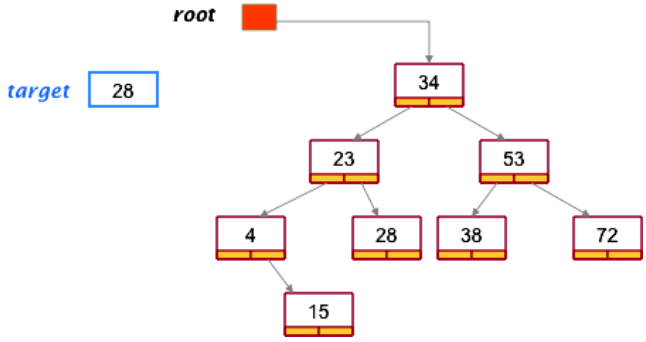
***Search for a Target Key* in a Binary Search Tree**

> ***Compare the Target Key with the Key of the Root Node***
>
> ***Equal: The Target is Found***
>
> ***Target Key is Smaller: Search for the Target Key in the Left Sub-Tree***
>
> ***Target Key is Larger: Search for the Target Key in the Right Sub-Tree***

Notes:

- The algorithm is recursive.
- All sub-trees of BSTs are also BSTs.
    - The problem of searching the whole BST can be broken down into two steps:
    - Step 1: Matching the current node
    - Step 2: Searching one of the sub-trees.

The following shows an example of searching for 28 in a BST.



Searching operation:

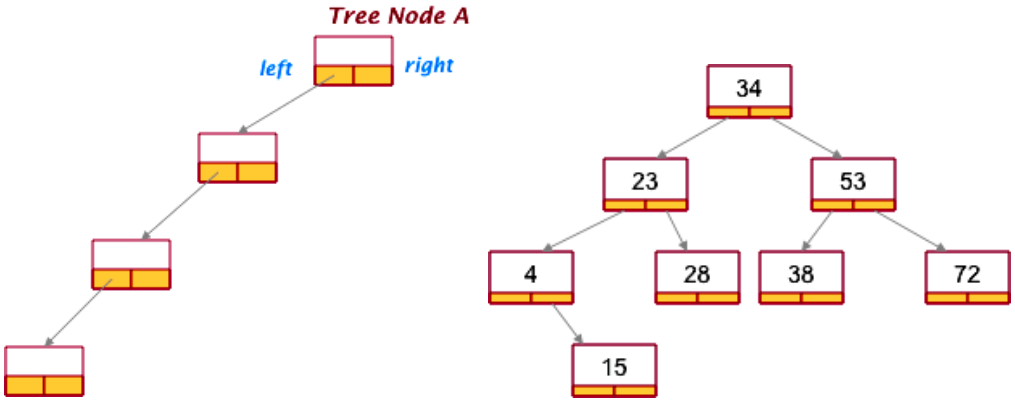
Step 1	Compare 28 to 34	Smaller: Search Left Sub-tree
Step 2	Compare 28 to 23	Larger: Search Right Sub-tree
Step 3	Compare 28 to 28	Equal: The Target is Found

Performance of Key Searching in BST

Key searching on BST can be very efficient.

- Every comparison can eliminate one of the sub-trees from the search space.
- Performance is similar to binary search
- However, actual performance is dependent on the tree shape.

Consider the following two BSTs.



Notes:

- The left tree is a degenerated tree.
- The performance of a degenerated tree is highly volatile and usually the worst case performance is poor.
    - The tree shape looks like a linked list, and the performance is similar to a linked list.
- The right tree is a balanced tree.
    - Key searching on balanced BST is most efficient.

Non-Key Searching in Binary Search Trees

Non-key searching, in the worst case, needs to traverse all nodes in the tree.

- No sub-tree elimination as happens in key searching.

The algorithm of non-key searching is shown below.

Algorithm: Non-Key searching in BST

```
Search for a Target Non-Key Data in a Binary Search Tree
        Compare the Target Data with the Data of the Root Node
        IF Equal: The Target is Found, RETURN Result
        ELSE
                Result = Search for the Non-Key in the Left Sub-Tree
                IF NOT FOUND
                        Result = Search for the Non-Key in the Right Sub-Tree
                        IF NOT FOUND
                                RETURN Not Found
                RETURN Result
```

Notes:

- The algorithm is recursive (again).
- All sub-trees of BSTs are also BSTs.
    - The problem of searching the whole BST can be broken down into three steps:
    - Step 1: Matching the current node
    - Step 2: Searching the left sub-tree
    - Step 3: Searching the right sub-tree

9.2.3 Data Insertion in Binary Search Trees

The data insertion operation in binary search trees is similar to the case of linked list:

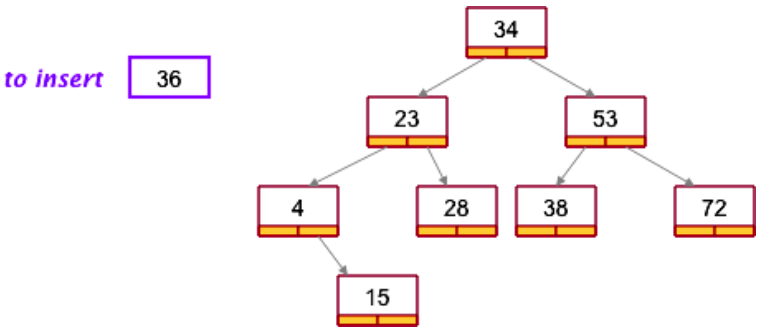
- Step 1: Create a new node
- Step 2: Link the new node to the structure

Binary search trees have imposed the golden rule on the ordering of nodes.

- New nodes cannot be added somewhere.
- They must be added at a location such that the new binary tree will remain a binary search tree.

The correct method of adding new nodes to binary search trees should follow:

- New nodes should be added as leaf nodes.
    - Existing nodes are not affected by the new nodes.
- The new node should be added at a position where a key searching operation would also reach.



Notes:

- If the target of key data searching is 36, the final location of the search operation would be the left child of node 38.
- This is the correct location of the new node 36.
- Any future key data searching operation would be able to find the node 36.

Recursive Algorithm for Data Insertion in BST

Data insertion in BST involves three steps:

- Step 1: Traversal based on key searching to find the correct location.
- Step 2: Create new node
- Step 3: Attach the new node at the location.

The following shows a recursive algorithm for data insertion in BST.

Algorithm: Data Insertion in BST (Recursive)

Insert a new node into a BST

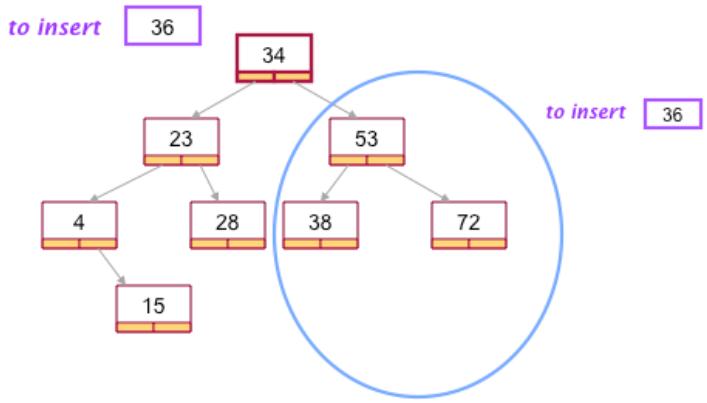
> ***Check if the root node is the right location (i.e. root node is None – an empty binary tree).***
>
> ***If not, then decide whether to go left branch or right branch.***
>
> ***Insert the new node to the left sub-tree, if the left branch is selected.***
>
> ***Insert the new node to the right sub-tree, if the right branch is selected.***

Notes:

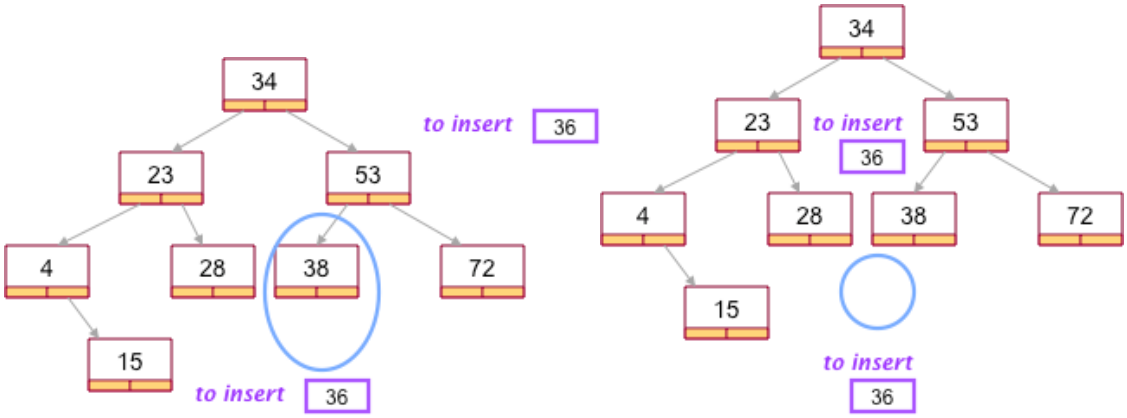
- Data insertion is essentially largely based on key searching.
  - Because key searching is recursive, data insertion should also be recursive.
- The candidate locations are the empty left child and right child of nodes.
  - The left link or the right link of a node is `None`.
- New node must be created at the location of *empty tree*.
  - The so-called empty trees are simply left or right links equal to `None`.

Example: Recursive Data Insertion in BST

Consider the example of inserting 36 to the BST. The root node is 34. Not the right place, 36 > 34, should insert to right sub-tree of 34.

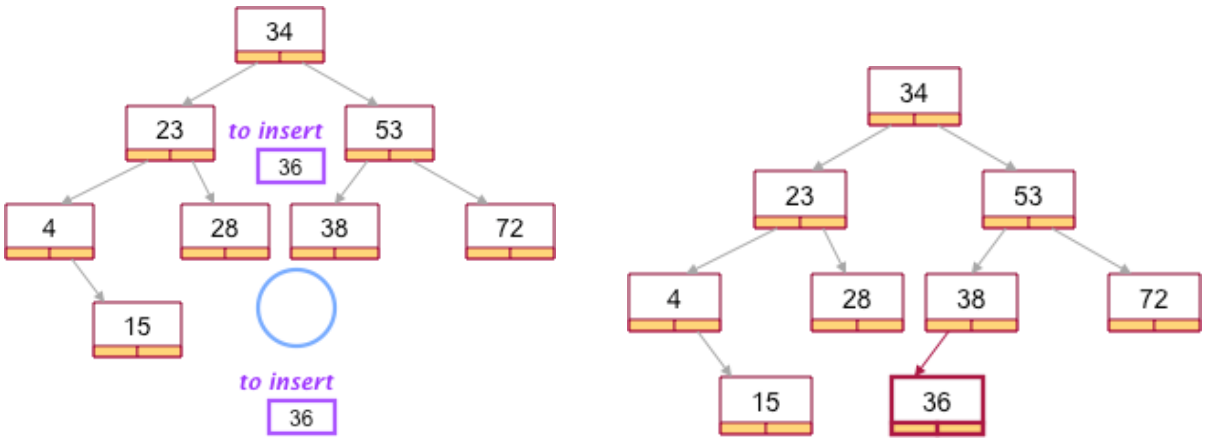


Consider the root node 53. Again, it is not the right place, 36 < 53, should insert to left sub-tree of 53.



Consider the root node 38. Again not the right place, 36 < 38, should insert to left sub-tree of 38.

The left sub-tree of 38 is empty. Therefore it is the correct location for attaching a new node.



9.2.4 Full Traversal in Binary Trees

Binary trees are non-linear structure, which means that each node can be linked to more than one node.

For binary trees, there are two types of traversal:

- Full traversal: meaning that all nodes should be visited.
- Key traversal: meaning that the nodes are visited according to the direction of the key.
    - Usually few nodes are visited in key traversal.

Traversal involves moving from one node to another.

- For linked lists, there are only two directions to move: backward and forward.
    - Once the direction is chosen, then there is only one route to traverse the linked list.
- For binary trees, there are a few routes to traverse due to the multiple links of every node.
- Traversal in binary trees always starts from the root node.

Full Traversal from Root Node

At the root node, there are three tasks to do in a full traversal.

- Visit the root node.
- Traverse the left-branch.
- Traverse the right-branch.

All three tasks have to be done in a full traversal. Then there are 3 choices of order of doing these tasks.

- Visit root node first, then traverse left, finally traversal right.
- Traverse left first, visit root node, finally traversal right.
- Traverse left first, then traversal right, finally visit root node

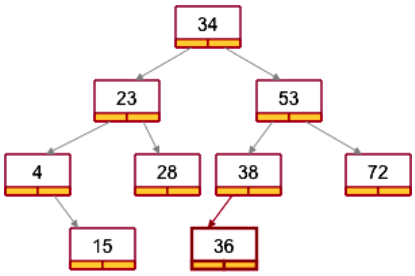
The above assumes forward traversal: always traverse left before traverse right.

If backward traversal is considered, then there are 6 choices.

Order of Traversal

If only forward traversal is considered, there are three orders of doing the 3 tasks at the root node.

Remind that BST is a recursive structure, and every node in a BST is a root node for a sub-tree. The same three choices are available when traversal reaches a node for the first time.



There are three major traversal orders in binary tree traversals. The traversal orders differ in the order of node and branch visit.

Pre-order traversal: root – left – right
In-order traversal: left – root – right
Post-order traversal: left – right – root

The **pre-order traversal** follows the order of root – left – right.

- At each node, the pre-order traversal would visit the node first (root),
- Then visit the left branch (left).
- After every node in the left branch is visited and returned to the node, the right branch (right) is visited.

A pre-order traversal on the above example tree visits nodes in the following order.

```
34 23 4 15 28 53 38 36 72
```

The **in-order traversal** follows the order of left – root – right.

- At each node, the in-order traversal would first visit the left branch (left).
    - The traversal has reached the node, but not visiting it yet.
- After the traversal finishes visiting the left subtree, now visit the node (root).
- Finally the right subtree (right) is visited.

An in-order traversal on the above example tree visits nodes in the following order.

```
4 15 23 28 34 36 38 53 72
```

Note:

- If the tree is a binary search tree, the in-order traversal will visit the nodes in order.

The **post-order traversal** follows the order of left – right – root.

- At each node, the in-order traversal would first visit the left branch (left).
    - The traversal has reached the node, but not visiting it yet.
- After the traversal finishes visiting the left subtree, visit the right subtree (right).
    - Again, the traversal has reached the node, but no visiting it yet.
- Finally the node is visited (root).

An post-order traversal on the above example tree visits nodes in the following order.

```
15 4 28 23 36 38 72 53 34
```

9.3 Implementation of Binary Search Trees

This section describes an implementation of binary search tree based on Python classes.

Two classes will be defined.

- The `Node` class
  - Defining the nodes for binary search tree.
  - Implemented recursive functions for tree computation
- The `BSTree` class.
  - Holds the reference to the root node of binary tree.
  - Offers interface for the methods provided in the Node class, with root node management. For example, checking if the tree is empty (i.e. the root node is `None`).
- The `BSTree` class is not necessary.
  - The methods in `BSTree` simple handover the work to the corresponding methods in the `Node` class.
  - The value added by `BSTree` is to handle the case of empty tree.
- If a programmer wishes to handle the empty tree case, only the `Node` class is needed.

The Definition of the Two Classes

Example: binary_tree_adt.py

```
class Node:
    def __init__(self, key, data=None):
        self.key = key    # the key for BST
        self.data = data  # the payload
        self.left = None
        self.right = None
   ...

class BSTree:
    def __init__(self):
        self.root = None  # link to root node
   ...
```

Notes:

- The Node class has the key and data part separated in two properties.
  - This implementation ensures that there is always a key for ordering the nodes.
  - The key should be an immutable value.
  - The data part can be any payload, mutable, immutable, and `None`.

9.3.1 Implementing The Node Class

All methods in the Node class are recursive.

- Binary trees are a recursive structure.
- Recursion can exploit the recursive structure with divide-and-conquer.
    - Each sub-tree of a binary search tree is also a binary search tree.
    - Any method written for a binary search tree is also applicable to any subtree within the binary search tree.

Key Searching and Data Insertion

These two operations are similar. They both involve key searching traversal.

The following shows an implementation of key searching.

Example: binary_tree_adt.py

```
...
    # searching for data based on key
    # return data
    def searchKey(self, key):
        if self.key is None:
            return None
        if key < self.key:
            if self.left is None:
                return None
            else:
                return self.left.searchKey(key)
        elif key > self.key:
            if self.right is None:
                return None
            else:
                return self.right.searchKey(key)
        else:
            return self.data  # the key is found
```

Notes:

- There are three stopping conditions for recursion.
    - If the key of any node is `None` (i.e. not defined), searching should stop. No key should be `None` and so the tree is erroneous.
    - If the left link is `None` when searching should go to the left subtree. No nodes remain to be search and so no result is found.
    - If the right link is `None` when searching should go to the right subtree. No nodes remain to be search and so no result is found.
- Recursive calls are made for searching the left sub-tree or searching the right sub-tree.
    - The sub-trees are also trees.
    - Searching sub-trees is a simplified version of the original problem of searching a tree.
    - The same method can be applied on the sub-trees.

The following shows an implementation of data insertion. It is very similar to the key searching.

Example: binary_tree_adt.py

```
...
    def insert(self, key, data=None):
        if self.key is None:
            return
        if key < self.key:
            if self.left is None:
                self.left = Node(key, data)
            else:
                self.left.insert(key, data)
        elif key > self.key:
            if self.right is None:
                self.right = Node(key, data)
            else:
                self.right.insert(key, data)
        else:
            self.data = data # replace the data if duplicated key
```

Traversals

The following shows an implementation of the three orders of traversals.

Example: binary_tree_adt.py

```
...
    # pre-order traversal
    def printPreOrder(self):
        print(self.key, self.data)
        if self.left:
            self.left.printPreOrder()
        if self.right:
            self.right.printPreOrder()

    # in-order traversal
    def printInOrder(self):
        if self.left:
            self.left.printInOrder()
        print(self.key, self.data)
        if self.right:
            self.right.printInOrder()

    # post-order traversal
    def printPostOrder(self):
        if self.left:
            self.left.printPostOrder()
        if self.right:
            self.right.printPostOrder()
        print(self.key, self.data)
```

Notes:

- The implementation is again recursive.
- The structure inside each of the function follows exactly as the order of the 3 tasks at each node.
    - Pre-order traversal should follow visit – left – right, which is exactly the same as the order of instruction in the function.
    - The printing of node content is considered as visiting the node.

Counting the Number of Nodes

The counting of nodes is simply a full traversal of nodes.

Example: binary_tree_adt.py

```
...
    # returns the number of nodes
    def countNodes(self):
        count = 1
        if self.left:
            count += self.left.countNodes()
        if self.right:
            count += self.right.countNodes()
        return count
```

Height of Trees

The height of a tree is also defined recursively.

The height of a tree is 1 + the height of its highest sub-tree. The height of an empty tree is undefined.

Example: binary_tree_adt.py

```
...
    # returns height of tree
    def treeHeight(self):
        leftheight = rightheight = 0
        if self.left:
            leftheight = self.left.treeHeight() + 1
        if self.right:
            rightheight = self.right.treeHeight() + 1
        return max(leftheight, rightheight)
```

9.3.2 Implementing The BinaryTree Class

The methods in the `BinaryTree` class deals with the possibility of the root node placeholder, and then hand over the calls to the `Node` class.

An implementation is shown below.

Example: binary_tree_adt.py

```
class BSTree:
    def __init__(self):
        self.root = None  # link to root node

    def insert(self, key, data=None):
        if self.root is None:
            self.root = Node(key, data)
        else:
            self.root.insert(key, data)

    # searching for data based on key
    # return data
    def searchKey(self, key):
        if self.root:
            return self.root.searchKey(key)
        return None

    def printPreOrder(self):
        if self.root:
            self.root.printPreOrder()

    def printInOrder(self):
        if self.root:
            self.root.printInOrder()

    def printPostOrder(self):
        if self.root:
            self.root.printPostOrder()

    # returns the number of nodes
    def countNodes(self):
        if self.root:
            return self.root.countNodes()
        return 0

    # returns height of tree
    def treeHeight(self):
        if self.root:
            return self.root.treeHeight()
        return None  # height is None for empty tree
```

9.3.3 Using The BinaryTree Class

To use the `BinaryTree` class, create an object of `BinaryTree` and call its method.

Example: binary_tree_adt.py

```
if __name__ == "__main__":

    tree = BSTree()
    numlist = [34, 23, 4, 15, 28, 53, 38, 36, 72]
    for num in numlist:  # add numbers to the binary tree
        tree.insert(num)

    print("Pre-order traversal")
    tree.printPreOrder()
    print("In-order traversal")
    tree.printInOrder()
    print("Post-order traversal")
    tree.printPostOrder()

    print("Number of nodes = ", tree.countNodes())
    print("Height of tree = ", tree.treeHeight())
```

```
Pre-order traversal
34 None
23 None
4 None
15 None
28 None
53 None
38 None
36 None
72 None
In-order traversal
4 None
15 None
23 None
28 None
34 None
36 None
38 None
53 None
72 None
Post-order traversal
15 None
4 None
28 None
23 None
36 None
38 None
72 None
53 None
34 None
Number of nodes =  9
Height of tree =  3
```

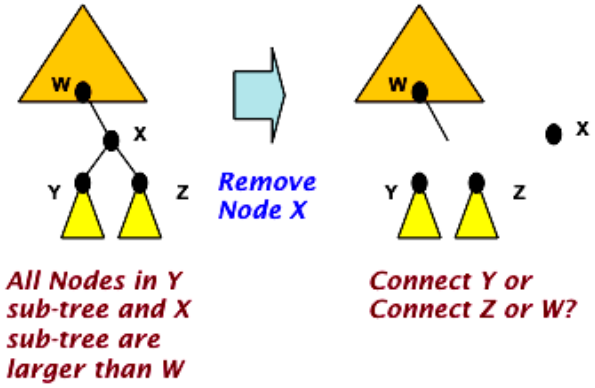
9.4 (Challenging) Node Deletion in Binary Search Trees

Node deletion is difficult to implement in binary search trees because of the complexity of pointer manipulation involved.

The difficulty of node deletion lies in maintaining the golden rule of a binary search tree.

- The nodes must obey the rule after a node is deleted.
- Deletion a node would leave a void to fill, if the node has a left sub-tree or a right sub-tree.
- The re-connection of the nodes must result in a binary search tree, where all nodes are ordered.

The following shows a general BST.



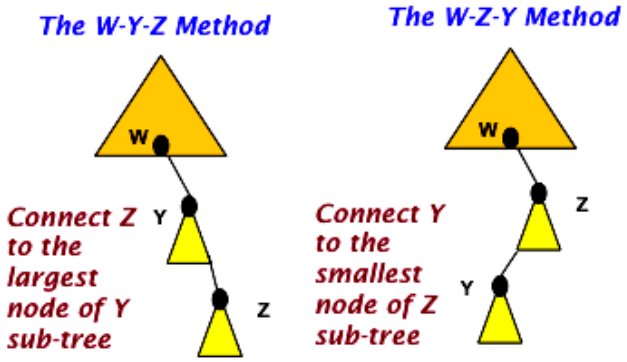
Notes:

- Somewhere in the tree there is a node X, which is to be deleted.
- Consider that X has two children node Y and node Z, and they are the roots of two sub-trees.
- If node X is removed, then there are two sub-trees to be connected back to the BST.

Reset the BST after Node Removal

There are two methods that will make the resulting tree a BST:

- The W-Y-Z method: connect the Z sub-tree to the largest node of Y sub-tree.
- The W-Z-Y method: connect the Y sub-tree to the smallest node of the Z sub-tree.



Notes:

- In the W-Y-Z method, the largest node in the Y sub-tree is next smaller to deleted node X.
  - The largest node in Y sub-tree is called the **immediate predecessor** of deleted node X.
- The smallest node in the Z sub-tree is the next larger to node X.
  - The smallest node in the Z sub-tree is called the **immediate successor** of deleted node X.

Based on the golden rule of BST, the following relation was found:

Largest node of Y sub-tree < node X < Smallest node of Z sub-tree

Any node of Y sub-tree is smaller than any node in Z sub-tree.

- Connecting Z sub-tree to the right branch of the largest node of Y preserves the order.
- Connecting Y sub-tree to the right branch of the smallest node of Z also preserves the order.

9.5 Characteristics of Binary Search Trees

This section describes the performance characteristics of binary search trees, and the gives a summary of the major properties.

9.5.1 Performance of Binary Search Trees

Common operations of binary search trees include data addition, key searching, non-key searching, traversal and data removal.

The time complexity classes for the major operations of binary search trees are given in the following table.

Operations	Average-case	Worst-case
Node addition	O(lg N)	O(N)
Key Searching	O(lg N)	O(N)
Non-Key Searching	O(N)	O(N)
Traversal	O(N)	O(N)
Node deletion	O(lg N)	O(N)

Notes:

- Node addition and key searching involves the same traversal process.
    - The process travels from the root node to a leaf node.
    - In a balanced tree, the height is lg (N), and this would be the distance travelled from the root node to a leaf node.
- The worst case performance happens with degenerate trees.
    - The performance is similar to that of a linked list.

9.5.2 Properties of Binary Search Trees

The main properties of binary search trees are described below.

Access to Elements

Binary trees, like linked lists, do not allow direct access to their data elements.

- Direct access is only allowed on the root node through the root pointer of the binary tree.
- In key searching, however, each node can serves as a signpost telling where the target node should be in the left branch or the right branch.
- Access to elements through **key searching** is significantly efficient than linked lists.

Memory Use

The memory is acquired dynamically as needed.

- When a binary tree node is deleted, the associated memory can be released immediately.
    - It ensures an efficient use of system resource.
- The drawback is of course the frequent memory allocation and de-allocation.
- The binary tree structure incurs marginally more memory overhead in having an additional pointer in the node.

Inserting and Deleting Elements

The insertion and deletion of data elements may be efficient.

- Both insertion and deletion involves traversal of nodes.
    - Their efficiency depends on the type of traversal required.
- The data insertion should be more efficient than linked list.
    - The keys of the nodes can point out the correct location for insertion.
- The performance of data deletion depends on how the target node is found.
    - If the target node is found through key searching, then deletion is more efficient than linked list.

The Capacity

The capacity of BST has no theoretical upper bound.

- The limit is the memory available in the operating systems.
- The capacity can grow or shrink in execution time, in accordance to the amount of data involved.

COMPS209F Data Structures, Algorithms and Problem Solving

Hashtables 10

Hashtable is an interesting data structure that provides constant time data insertion and retrieval.

- The data to store is in the form of key-value pairs.
- It is very efficient in data insertion and data retrieval.
- It costs more in memory size.
    - It requires a large memory block for its operation.
    - The performance would degrade if the memory block is not sufficiently large.

Python provides hashtables, in the form of dictionaries. Python's `dict` class is actually hashtable and this class can offer very efficient data insertion and data retrieval.

This chapter describes the inner working of hashtables.

10.1 Introduction to Hashtables

Data searching and retrieval with hashtable is usually faster than binary search.

Binary search is already a very efficient data search algorithm.

- The worst case time complexity is O(lg N), where N is the number of data items.
- The weakness of binary search is a requirement concerning the ordering of data.
- Sorting is an expensive operation.
- Unless the data set is absolutely stable, otherwise a sorting operation is required after each alteration to the data set.

Hashtables is a data structure based on the search algorithm of **hashing**.

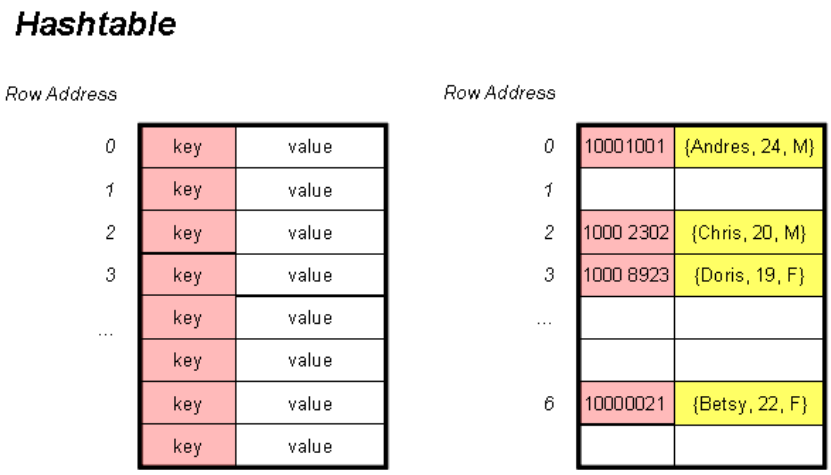
- A hashtable contains rows of data, and each row contains a key part and a data part.
- Hashing is a process that maps a key to a row address of the table.
- This process is applied when the data is inserted into the table and during a search for data with the key.

Hashing may be regarded as a magical formula that calculates the ideal location for storing a data in the table. However, it is not magical. It is based on understanding of how hashing works.

10.1.1 Hashing and Hashtables

The following figure shows an example of a hash table. A hash table has many rows, each of which is a (key, value) pair. A row can be empty. The number of rows is fixed.

The key part is student ID, and the value or data part contains the name, age, and sex of a student.



Hashing

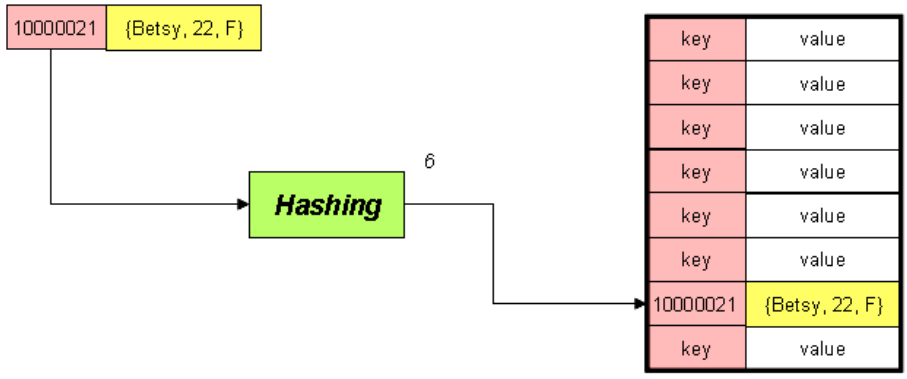
Hashing is a function that accepts one value (the key) and produces another value (the address).

- The same hashing function always produces the same address for the same key.
- The same key is mapped to the same address during data insertion and during data search.
- The address refers to the row ID of the table

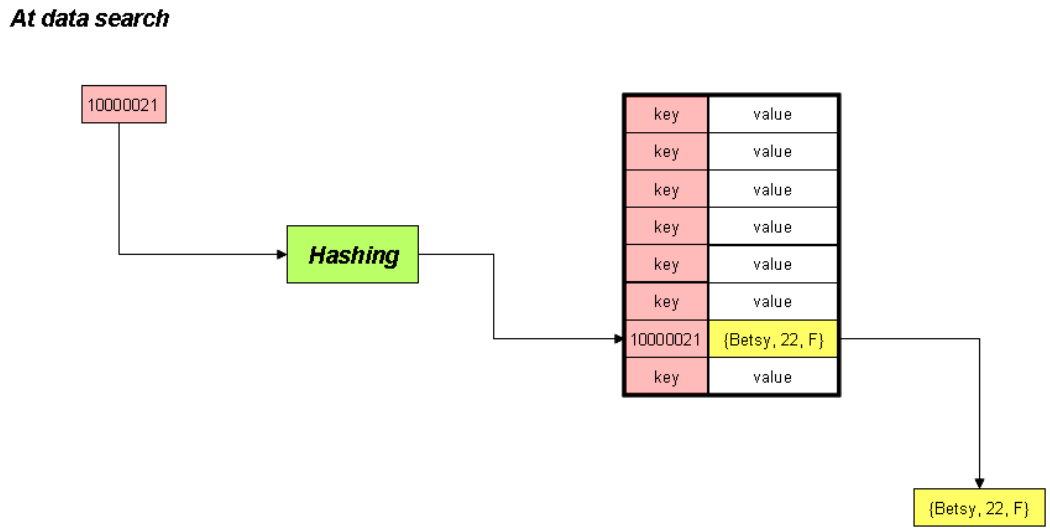
The following shows an example of inserting student data for ID 10000021 with the data {Betsy, 22, F}.

- The hash function produces the address 6 given the ID 10000021.
- The data is to be inserted at row address 6.

At data insertion



At data search stage, if a process wants to find out the data associated with ID 10000021, then the same hash function is applied and the row address 6 is produced. The data can be found at row address 6 in the table.



Performance of Data Search with Hashtable

Hash table is therefore incredibly fast in data search. The execution of the hash function is more or less constant time. So the desired data is found with a straightforward calculation of the row address only. The time complexity for data retrieval is constant time.

Hash table is a classic example of **space-time trade-off**.

- The fast data retrieval comes at a cost of memory space.
- The table size must be sufficiently large so that any key could be mapped to a unique row address.
- Collision means that two keys are mapped to the same row address. This is a possibility for a table of finite size.
- A large table reduces the possibility of collision, but it consumes more memory space.

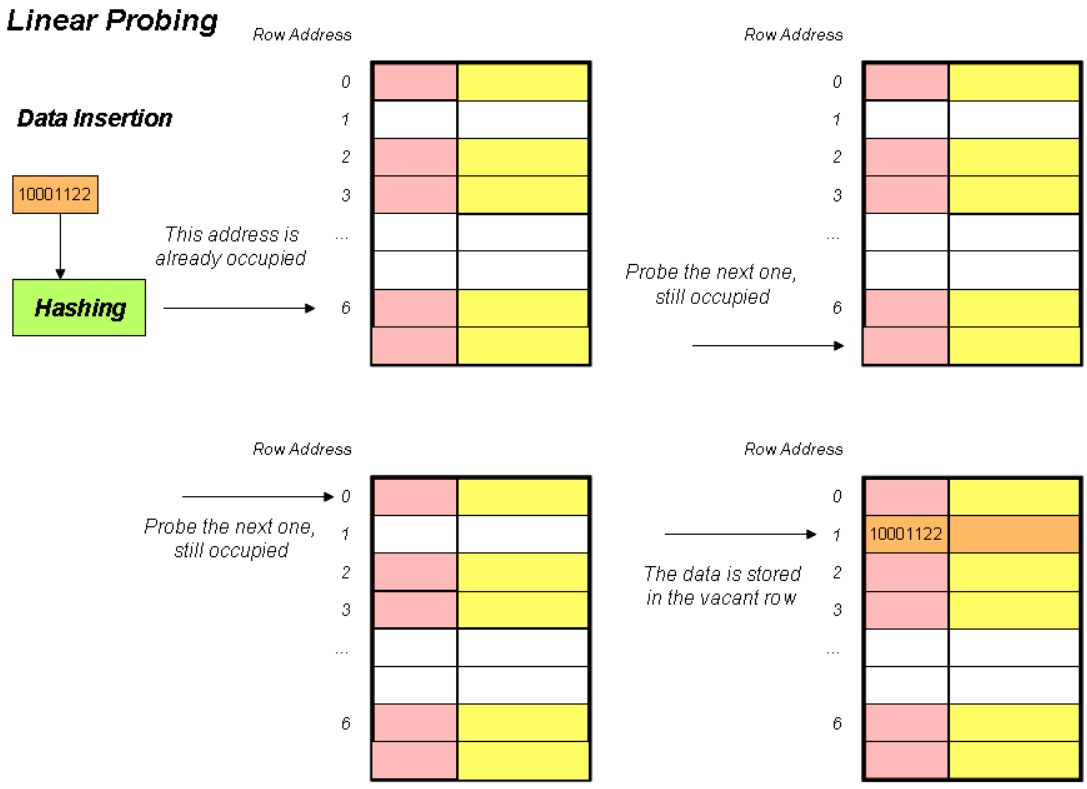
10.1.2 Collision Resolution

Any implementation of hash table is likely to be of a finite capacity, and therefore there is always a possibility of collision. When two keys are mapped to the same row address, then something must be done.

The following shows a situation of collision.

- A new student with ID 10001122 is to be added to the hash table.
- The row address generated is 6, which happens to be occupied.
    - The location occupied means that another ID also mapped to the address 6 by the hash function.
- In this case, there is no place to insert the new data. There are two solutions
    - Solution 1: print an error message. This is not a useful solution.
    - Solution 2: resolve the collision.

Linear Probing



Linear probing is one common method of collision resolution.

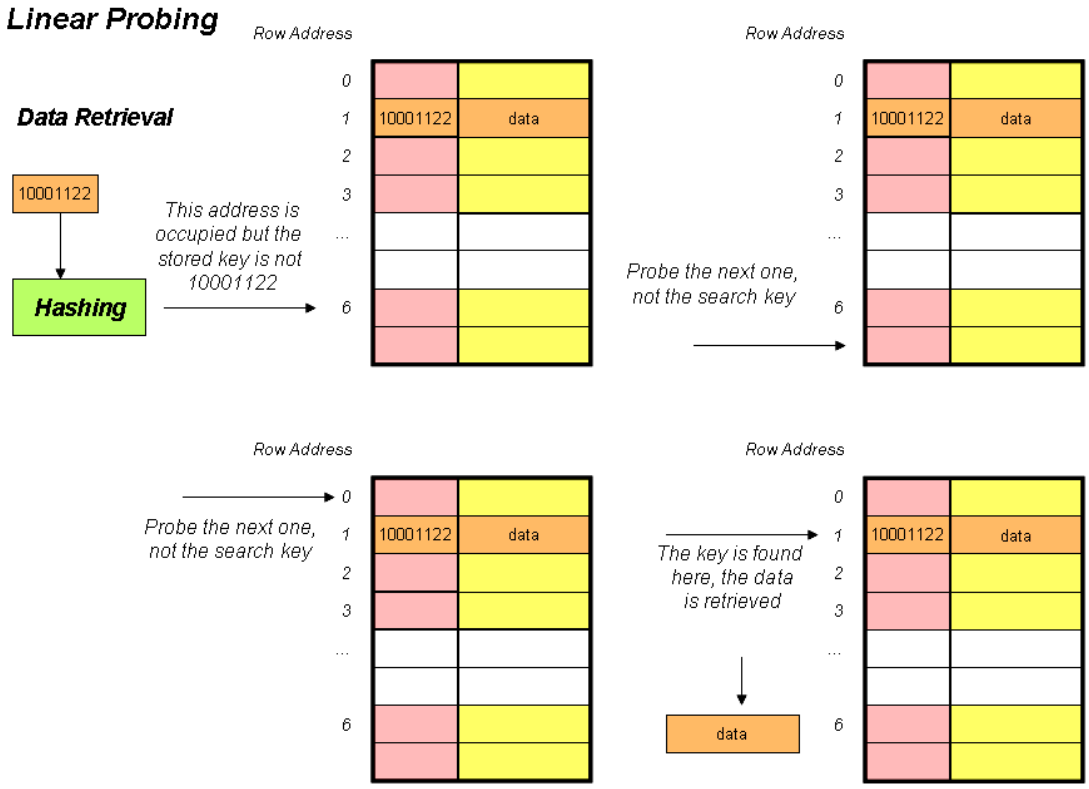
- The idea of linear probing is to start a searching process.
- The process moves to the next row if the current row is already occupied (i.e. a collision).
- It keeps looking until a vacant row is found.
- It stops when there is no vacant row, which means that the hash table is full.
- If it reaches the end of the table, the first row is checked.

The above figure show how linear probing is used to resolve collision and find the next available vacant row for the new data.

Any collision resolution method is not useful if the data cannot be retrieved.

- With linear probing, the data can always be retrieved by repeating the same process.
- If the hash function produced an address that is occupied but the key is not the search key, then linear probing is applied.
- The next rows are checked one by one until either one of the following conditions:
    - The desired search key is found. The associated data is retrieved
    - A vacant row. There is no data associated with the given key.

The following figure shows how to retrieve data through hashing and linear probing.



Linear probing is a simple but effective method for collision resolution. It has a number of disadvantages such as the data tended to cluster and the consequential rapid degradation of efficiency. Other methods include the following:

- Double hashing: Another hash function is consulted when there is a collision. This hash function's result is added to the original row address. The result is usually away from the first spot and avoiding clustering of data.
- Chaining. Add a dynamic data structure such as a linked list at each row. In the case of collision, the new data is added to the end of the linked list associated with the row.

10.2 Performance of Hashtables

The performance analysis of hash table is difficult because the performance is very sensitive to the implementation details and the operation parameters

There are a number of implementation factors that would affect the performance:

- Hash function: the better hash function is able to distribute different input data to different addresses.
- Table size: the larger the table size, the less likely collision would occur. Dynamic resizing could be used to enlarge the table size.
- Collision resolution algorithm: linear probing, double hashing, and chaining methods would result in different types of performance with different input data characteristics.

The time complexity classes for the major operations of hashtables are given in the following table.

Operations	Average-case	Worst-case
Key Searching	O(1)	O(N)
Non-Key Searching	O(N)	O(N)
Insert	O(1)	O(N)
Delete	O(1)	O(N)

Notes:

- Key searching, data insertion, and data deletion require the location of the data.
    - The hash function calculates the location of data.
    - No traversal is needed and so the time is independent of the data size.
- In the worst case, linear probing or other collision resolution process would need to traverse through all data.
    - The time depends linearly on the data size.

References

The design of hash function is important for the hash table to operate efficiently. The hash function should ideally distribute the row addresses evenly so that the data will not cluster together.

The study of hash functions is an advanced topic. For those who are interested, you can study the following articles.

Hash function at Wikipedia

http://en.wikipedia.org/wiki/Hash_function

Hashing Tutorial

http://research.cs.vt.edu/AVresearch/hashing/

Hashing Animation Tool

http://groups.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm

Hashing Animation Tool Help

http://groups.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashHelp.htm