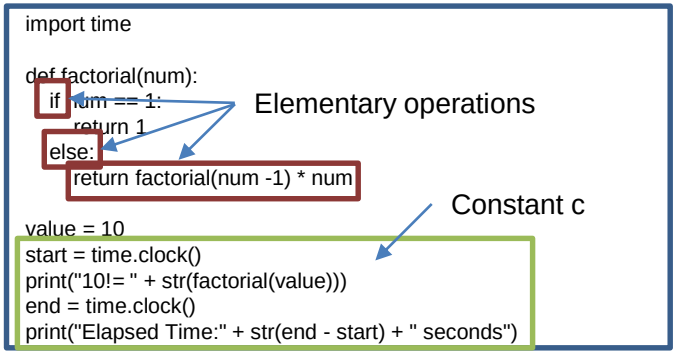
SIT32004
ICT Application Development

Lecture 02

Python Overview

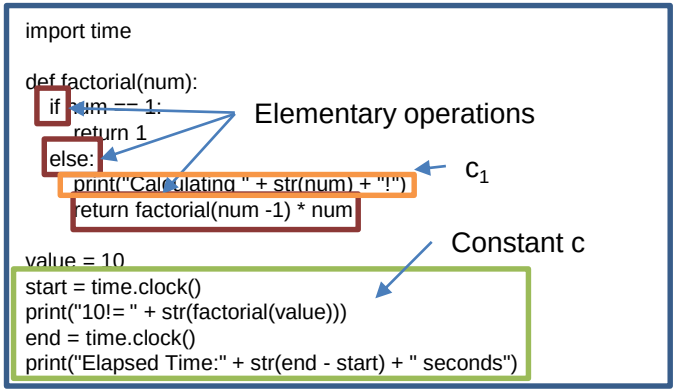
How to improve the performance of your app? (1/2)

- Definition of performance
  - Time complexity
    - » Time complexity is commonly **estimated** by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform [1]
  - In practical,
    - » Measure the CPU Time
    - » How many instructions are needed to solve a problem?



Time Complexity: O(n)
# of elementary operations called: n
# of operations called: 2*n + c

Time consumption: 0.0001524 seconds



Time Complexity: O(n)
# of elementary operations called: n
# of operations called: $(c_1+2)*n + c$

Time consumption: 0.0014052 seconds

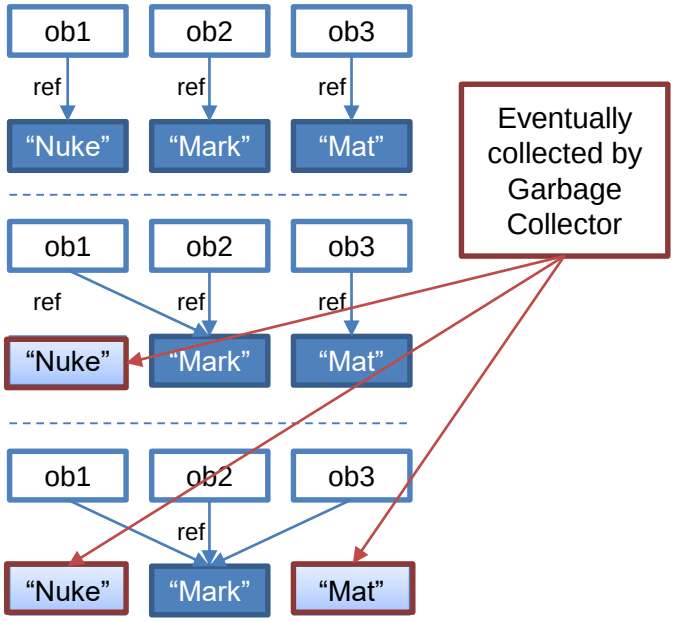
How to improve the performance of your app? (2/2)

- Definition of performance
  - Memory
    - » Space complexity
    - » How much space does it need to solve a problem?
  - In practical
    - » Measure the memory usage, and not to make “memory leak”
    - » Memory leak
      - Memory that was used once, and now is not, but has not been reclaimed
      - Not possible when you use pure Python code (Garbage Collection)

```
def print_obj(lst):
    print("---")
    for ob in lst:
        print(ob)

class Student(object):
    def __init__(self, string):
        self.name = string

ob1 = Student("Nuke")
ob2 = Student("Mark")
ob3 = Student("Mat")

print_obj((ob1, ob2, ob3))
ob1 = ob2
print_obj((ob1, ob2, ob3))
ob3 = ob1
print_obj((ob1, ob2, ob3))
```

```
---
<__main__.Student object at 0x01117250>
<__main__.Student object at 0x011172D0>
<__main__.Student object at 0x011172F0>
---
<__main__.Student object at 0x011172D0>
<__main__.Student object at 0x011172D0>
<__main__.Student object at 0x011172F0>
---
<__main__.Student object at 0x011172D0>
<__main__.Student object at 0x011172D0>
<__main__.Student object at 0x011172D0>
```

Execution Results



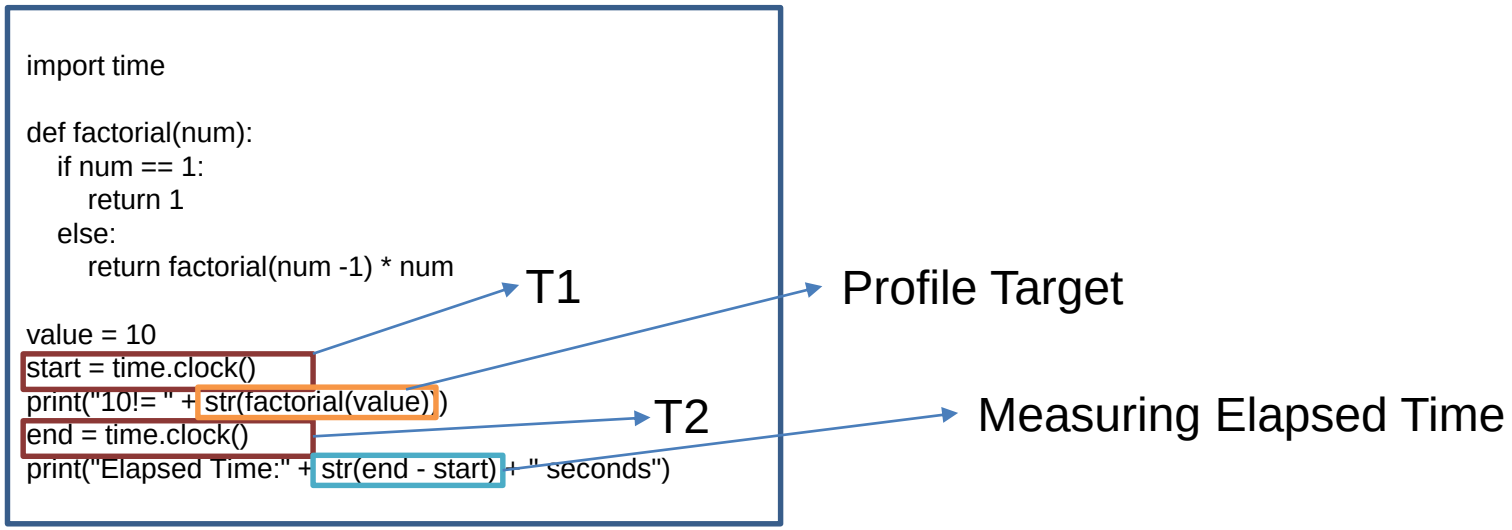
Profiling

- Definition
  - Profiling is a form of **dynamic program analysis** that **measures** the **usage of particular instructions**, or the **frequency** and **duration of function calls**[2].
  - Profiling is achieved by **instrumenting** either the program source code or its binary executable form using a tool called a profiler (or code profiler).

- Python Profiling Technique
  - Time
    - » %timeit command in IPython
    - » time.time() or time.clock()
    - » Timing decorator
    - » cProfile
    - » line_profiler
  - Memory
    - » heapy
    - » dowser
    - » memory_profiler

How to measure time? (1/3)

- print() and time.clock()
  - You may use time.clock() to measure time consumption.
  - How it works
    - » You should measure time before the execution of a function
      - Let it be T1
    - » Execute the function
    - » Measure the time after the function
      - Let it be T2
    - » Subtract T2 – T1



How to measure time? (2/3)

- Decorators [3]
  - Decorators are functions which modify the functionality of other functions
    - » May give a function as an argument to another function

```
from functools import wraps

def decorator(fn):
    @wraps(fn)
    def wrapFunction(*args, **kwargs):
        print("Invokation before function " + fn.__name__)
        fn(*args, **kwargs)
        print("Invokation after function " + fn.__name__)
    return wrapFunction

@decorator
def my_func():
    print("I am a function")

my_func()
```

For multiple decorator

To pass arguments to the Given function

Applying decorator to my_func

Execution Results

```
Invokation before function my_func
I am a function
Invokation after function my_func
```

  - Timing Decorator

```
def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.perf_counter()
        result = fn(*args, **kwargs)
        t2 = time.perf_counter()
        print("@timefn: {} took {} seconds".format(fn.__name__, t2 - t1))
        return result
    return measure_time
```

Measure Time

How to measure time? (3/3)

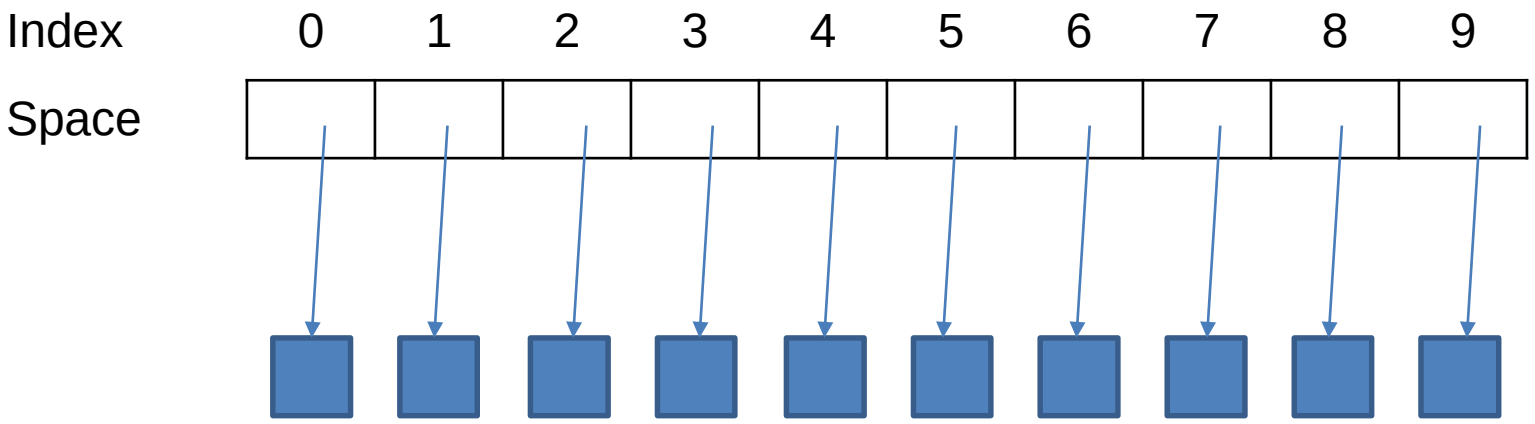
- line_profiler
  - The line_profiler is an open source software that profiles given a python program and analyze the execution results
  - You may clone project from https://github.com/rkern/line_profiler
    - » First you should clone the project
    - » Run following code at your command line with administrator privilage
      - python setup.py install
  - How to use the profiler
    - » Decorate functions which you want to analyze with @profile decorator
    - » Execute following command
      - kernprof -l -v {your program}

```
@profile
def factorial(num):
    if num == 1:
        return 1
    else:
        print("Calculating " + str(num) + "!")
        return factorial(num -1) * num

@profile
def profiling_factorial():
    value = 10
    result = factorial(value)
    print("10!= " + str(result))

if __name__ == "__main__":
    profiling_factorial()
```

```
PS C:¥Users¥cbchoi¥source¥repos¥Example¥Example> kernprof -l -v .¥Lecture02_Profiling03.py
Caluating 10!
Caluating 9!
Caluating 8!
Caluating 7!
Caluating 6!
Caluating 5!
Caluating 4!
Caluating 3!
Caluating 2!
10!= 3628800
Wrote profile results to Lecture02_Profiling03.py.lprof
Timer unit: 1e-07 s

Total time: 0.007939 s
File: .¥Lecture02_Profiling03.py
Function: factorial at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def factorial(num):
     5        10        214.0     21.4      0.3      if num == 1:
     6         1          9.0      9.0      0.0          return 1
     7                                               else:
     8         9      78046.0   8671.8     98.3          print("Caluating " + str(num) + "!")
     9         9       1121.0    124.6      1.4          return factorial(num -1) * num

Total time: 0.0106828 s
File: .¥Lecture02_Profiling03.py
Function: profiling_factorial at line 11

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    11                                           @profile
    12                                           def profiling_factorial():
    13         1         99.0     99.0      0.1      value = 10
    14         1      80157.0  80157.0     75.0      result = factorial(value)
    15         1      26572.0  26572.0     24.9      print("10!= " + str(result))
```

List & Tuples

- Each list and tuples is an array structure
  - Types
    - » Dynamic Array: List
    - » Static Array: Tuple
  - Characteristics
    - » Access elements by subscript(index)
    - » O(1) to access data



- » What about time complexity of inserting data to the data structure?

List

- Characteristics of the List structure
  - List is a collection of arbitrary objects
    - » It may contain various types of objects
  - List is ordered data structure
    - » The order of the elements does not change
  - Elements of the list can be accessed by index
    - » You may access the data with [ ] operator and the index
  - List is mutable
    - » You may change the contents of the element
  - List is dynamic
    - » You may add an element and remove an element
- Performance of the List[4]
  - Insertion: O(n)
  - Deletion: O(n)
  - Iteration: O(n)
  - Get Item: O(1)
  - Set Item: O(1)

Tuples

- Characteristics of the Tuple structure
  - Tuple is a collection of arbitrary objects
    - » It may contain various types of objects
  - Tuple is ordered data structure
    - » The order of the elements does not change
  - Elements of the tuple can be accessed by index
    - » You may access the data with [ ] operator and the index
  - **Tuple is immutable**
    - » You cannot change the contents of the element

```
t1 = (1, 2)
t1[1] = 10
print(t1)
```

Not allowed

- Performance of the Tuple
  - Insertion: N/A
  - Deletion: N/A
  - Iteration: O(n)
  - Get Item: O(1)
  - Set Item: N/A

```
t1 = ([1], [1])
print(t1)

t1[1][0] = 10
print(t1)
```

Allowed

Why this is allowed? (Homework)

Analysis of the List & Tuple

- Time analysis of iteration
  - Linear Search
    - » To unsorted data structure

```
def linear_search(lst, number):
    for i in range(len(lst)):
        if lst[i] == number:
            return True
        else:
            pass
    return False
```

  - Binary Search
    - » To sorted data structure

```
def binary_search(lst, number):
    if len(lst) == 0:
        return False
    else:
        mid_index = int(len(lst)/2)
        mid = lst[mid_index]

        if number < mid:
            return binary_search(lst[:mid_index], number)
        elif number > mid:
            return binary_search(lst[mid_index + 1:], number)
        elif number == mid:
            return True
        else:
            return False
```

Improve Performance of Iteration

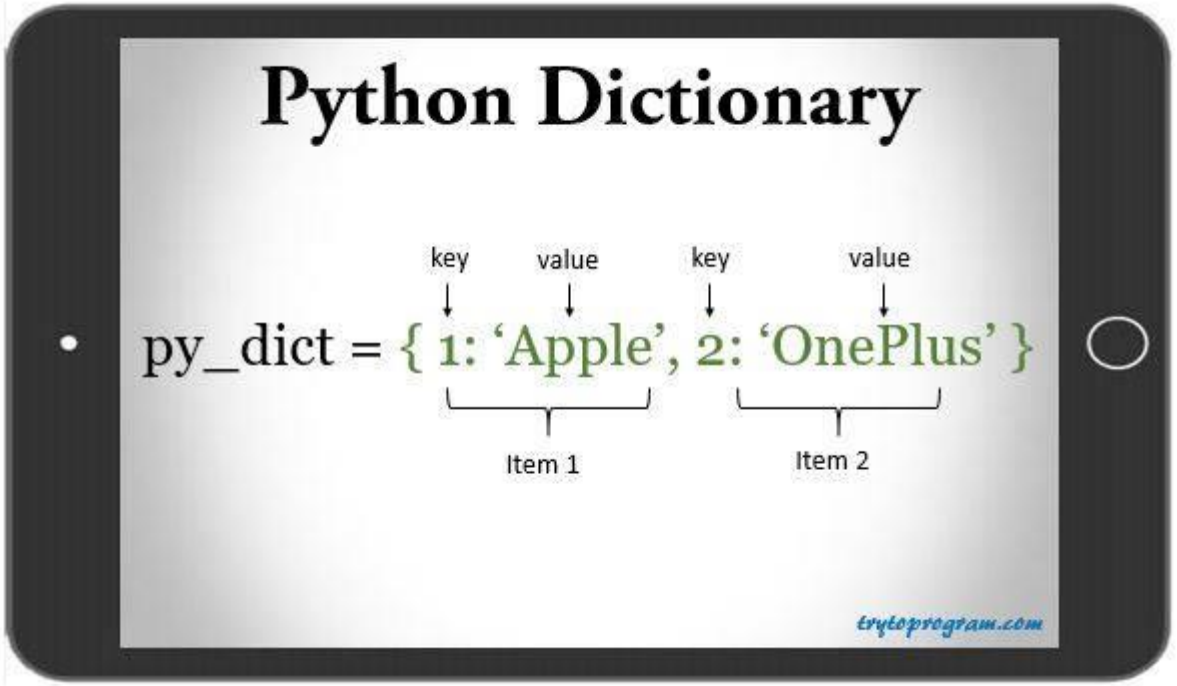
- Keep “Sorted Structure”
  - Apply sort() member function of the list every time when a user append an element
  - Using bisect [5]
    - » Array bisection algorithm
      - This module provides support for maintaining a list in sorted order without having to sort the list after each insertion.

```
@timefn
def insertion01():
    important_data = []
    for i in range(10000):
        new_number = random.randint(0, 1000)
        important_data.append(new_number)
        important_data.sort()

@timefn
def insertion02():
    important_data = []
    for i in range(10000):
        new_number = random.randint(0, 1000)
        bisect.insort(important_data, new_number)

insertion01()
insertion02()
```

```
@timefn: insertion01 took 0.33960060000000003 seconds
@timefn: insertion02 took 0.030492399999999975 seconds
```

Dictionary & Set

- Dictionary is an associative array
  - A dictionary consists of a collection of key-value pairs
  - Each key-value pair maps the key to its associated value



Referenced from http://www.trytoprogram.com/python-programming/python-dictionary/

Dictionary

- Characteristics of Dictionary
  - Dictionary is a collection of arbitrary objects
    - » It may contain various types of objects
  - Elements of the dictionary can be accessed by keyword
    - » You may access the data with [ ] operator and the keyword
  - Dictionary is mutable
    - » You may change the contents of the element
  - Dictionary is dynamic
    - » You may add an element and remove an element

- Performance of the Dictionary[4]
  - Insertion: O(n)
  - Deletion: O(1)
  - Iteration: O(n)
  - Get Item: O(1)
  - Set Item: O(1)

Set

- Characteristics of Set
  - Sets are unordered
  - Set elements are unique
    - » Duplicate elements are not allowed
  - A set itself may be modified
    - » However, the elements contained in the set must be an immutable type

- Performance of the Set[4]
  - x in s: O(1)
  - Union s|t: O(len(s) + len(t))
  - Intersection s&t: O(min(len(s), len(t))
  - Difference s-t: O(len(s))

Class, Object, Instance

- Python Class
  - Python is an object oriented programming language.
  - Almost everything in Python is an object, with its properties and methods.
  - A Class is like an object constructor, or a **"blueprint" for creating objects**.

- Object
  - Generally an object corresponds to **some real-world entity** in the problem space
  - In computer science, an object can be a variable, a data structure, a function, or a method, and as such, is a value in memory referenced by an identifier.
  - In the class-based object-oriented programming paradigm, object refers to a particular **instance of a class**, where the object can be a combination of variables, functions, and data structures.

Types of Objects

- Entity Object
  - Contain properties about themselves
  - Properties of the object can be modifiable through certain rules
- Control Object (Manager Objects)
  - Responsible for the coordination of other objects
  - Control and make use of other objects
- Boundary Object
  - Any object which takes input from or produces output to another system
  - Responsible for translating information into and out of the system
- Class and Architecture Design
  - Next week

How to use Open Source in Python?

1. You should get familiar with git
   1. Go to github.com and search for open source software
   2. You should understand how to use the following commands
      - » git clone https://{url}
      - » git commit -m “{commit log}”
      - » git push
      - » git pull
      - » git merge

2. You should read README.md file
   - There are useful information inside of README.md file
     - » Installation
     - » Usage
     - » Package Dependency

3. You should understand how to integrate software into your project
   - Integration is not a easy job. You should practice a lot

Reference

- [1] Time Complexity, available from https://en.wikipedia.org/wiki/Time_complexity
- [2] Profiling, available from https://en.wikipedia.org/wiki/Profiling_(computer_programming)
- [3] Decorator, available from http://book.pythontips.com/en/latest/decorators.html
- [4] Time Complexity of List in Python, available from https://wiki.python.org/moin/TimeComplexity
- [5] Bisection Algorithm, available from https://docs.python.org/3/library/bisect.html