

# CS1010S Tutorial 8

Wang Zexin

National University of Singapore  
Year 2 QF

*wang.zexin@u.nus.edu*

March 26, 2017

# Today's Agenda

- 1 Recap
- 2 Question One
  - Part A
  - Part B
  - Part C
- 3 Question 2
  - Standard solutions
- 4 Question 3
  - Part A
  - Part B
- 5 Question 4
- 6 Question 5
  - Part A
  - Part B
- 7 Extra stuff: A simple final question on Dictionary

# Recap - Message Passing & Stateful Objects

- A function
- A higher order function
- A higher order function object
  - can accept parameters like message
  - and other number also
- What you want is basically
- $A = \text{make\_something}()$
- $A(\text{'add'}, 10)$
- $A(\text{'subtract'}, 20)$
- A recipe of making such object
  - An internal storage object outside the helper function
  - If - elif - else statement to check message
  - return a helper function

# Recap - Message Passing & Stateful Objects

A more in-depth explanation using example.  
Where are all the ingredients for the recipe?

```
def make_widget():  
    stuff = ["empty", "empty", 0]  
    def oplookup(msg,*args):  
        if msg == "insert":  
            place = stuff[2]  
            stuff[place] = args[0]  
            stuff[2] = (place + 1) % 2  
        elif msg == "retrieve":  
            return stuff[stuff[2]]  
        else:  
            raise Exception("widget doesn't " + msg)  
    return oplookup
```

# Recap - VarArgs

A technique to take unknown number of arguments

- Use '\*' to denote the input parameter
- Removing '\*' will give a list of the arguments
- Use '\*' against when passing these arguments to another function

# Recap - Dictionary

## A new data structure

- keys and values - iterable objects in **order**
- construct a dictionary
  - `dict([(1,2),(2,4),(3,6)])`
  - `dict()`
  - `{'a':1, 'b':2}`
- check for a key in dictionary:
  - `d.get(key)` - None if no such key
  - `if d[key]:` - may have key error
  - `if key in d:`
  - `if key in d.keys():`
- `del d[key]`
- `d.clear()`

## Question 1 Part A

Describe in simple English how a widget behaves. Your explanation should be comprehensible to a layman who does not understand programming.

```
def make_widget():
    stuff = ["empty", "empty", 0]
    def oplookup(msg,*args):
        if msg == "insert":
            place = stuff[2]
            stuff[place] = args[0]
            stuff[2] = (place + 1) % 2
        elif msg == "retrieve":
            return stuff[stuff[2]]
        else:
            raise Exception("widget doesn't " + msg)
    return oplookup
```

# Question 1 Part A Discussion

How this widget behaves?

- A widget is made by *make\_widget()*
- It can only store a maximum number of 2 objects inside.
- You can use a statement to insert: *w('insert', object1)*
- You can use a statement to retrieve : *w('retrieve')*
- When you try to insert into an already full widget, the one inserted first will be removed.

```
def make_widget():
    stuff = ["empty", "empty", 0]
    def oplookup(msg,*args):
        if msg == "insert":
            place = stuff[2]
            stuff[place] = args[0]
            stuff[2] = (place + 1) % 2
        elif msg == "retrieve":
            return stuff[stuff[2]]
        else:
            raise Exception("widget doesn't " + msg)
    return oplookup
```



## Question 1 Part B

Write the code required to insert the following objects into the widget: 1, 2, 3.

- `w = make_widget()`
- `w('insert', 3)`
- `w('insert', 2)`
- `w('insert', 1)`

```
def make_widget():  
    stuff = ["empty", "empty", 0]  
    def oplookup(msg,*args):  
        if msg == "insert":  
            place = stuff[2]  
            stuff[place] = args[0]  
            stuff[2] = (place + 1) % 2  
        elif msg == "retrieve":  
            return stuff[stuff[2]]  
        else:  
            raise Exception("widget doesn't " + msg)  
    return oplookup
```

## Question 1 Part C

Suppose we perform a retrieval 3 times. What is returned in each time?

- w('retrieve')
- w('retrieve')
- w('retrieve')

```
def make_widget():  
    stuff = ["empty", "empty", 0]  
    def oplookup(msg,*args):  
        if msg == "insert":  
            place = stuff[2]  
            stuff[place] = args[0]  
            stuff[2] = (place + 1) % 2  
        elif msg == "retrieve":  
            return stuff[stuff[2]]  
        else:  
            raise Exception("widget doesn't " + msg)  
    return oplookup
```

Does the value of the widget ever change?

## Question 2

An accumulator is a function that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a function *make\_accumulator* that generates accumulators, each maintaining an independent sum.

## Question 2 Discussion

Your *make\_accumulator* should return:

- A function
- which manages an internal value
- stored in an internal object
- with its internal value changes
- everytime it is called

## Question 2 Zexin's solution

```
def make_accumulator():  
    s = [0]  
    def accumulate(number):  
        s[0] += number  
        return s[0]  
    return accumulate
```

Note that for the recipe:

- *s* is the internal storage object
- *accumulate* is the helper function returned
- and there is no if statement to check the message

## Question 3 Part A

Write a function *make\_monitored* that takes as input a function, *f*, that itself takes one input. The result returned by *make\_monitored* is a third function, say *mf*, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to *mf* is the special string 'how-many-calls?', then *mf* returns the value of the counter. If the input is the special string 'reset-count', then *mf* resets the counter to zero. For any other input, *mf* returns the result of calling *f* on that input and increments the counter.

## Question 3 Part A Discussion

Your object should do these:

- Take note of how many times  $f$  is called
- Return the number of calls when 'how-many-calls' is entered
- Set the counter in internal object to 0 when 'reset-count' is entered
- Otherwise, pass the input to  $f$
- Assume that there is only one input for  $f$ !

## Question 3 Part A Zexin's Solution

```
def make_monitored(f):  
    counter = [0]  
    def monitor(m):  
        if m == "how-many-calls?":  
            return counter[0]  
        elif m == "reset-count":  
            counter[0] = 0  
        else:  
            counter[0] += 1  
            return f(m)  
    return monitor
```



## Question 3 Part B

Extend *make\_monitored* so that it works for functions that take an arbitrary number of parameters.

## Question 3 Part B Discussion

The difficulty lies in that

- You need to know how to take in VarArgs
- You need to know how to pass VarArgs to  $f$
- Does VarArgs work for the case when there is no parameter?
- What if there is no parameter passed in?

## Question 3 Part B Zexin's Solution

```
def make_monitored_extend(f):  
    counter = [0]  
    def monitor(*m):  
        if len(m) == 0:  
            counter[0] += 1  
            return f()  
        elif m[0] == "how-many-calls":  
            return counter[0]  
        elif m[0] == "reset-count":  
            counter[0] = 0  
        else:  
            counter[0] += 1  
            return f(*m)  
    return monitor
```

## Question 4

Implement Monte Carlo integration as a function *make\_monte\_carlo\_integral* that takes as arguments a predicate  $P$ , lower and upper bounds  $x1$ ,  $y1$ ,  $x2$ , and  $y2$  for the rectangle and returns a new function.

## Question 4 Discussion

How to tackle this?

- Listen to Zexin on what is Monte Carlo integration first
- You will need to record down the number of success and trials
- You will calculate the probability of the random point being  $P$
- You will estimate the area using the total area and probability
- Is Monte Carlo integration unbiased? Try to prove it

## Question 4 Zexin's solution

```
def make_monte_carlo_integral(P, x1, y1, x2, y2):  
    counter = [0, 0]  
    area = abs(x2 - x1)*abs(y2 - y1)  
    def monte_carlo_integral(msg, *args):  
        if message == "run trials":  
            counter[1] += args[0]  
            for i in range(args[0]):  
                x = random.uniform(x1, x2)  
                y = random.uniform(y1, y2)  
                if P(x, y):  
                    counter[0] += 1  
        elif message == "trials":  
            return counter[1]  
        elif message == "get estimate":  
            return area * counter[0] / counter[1]  
    return monte_carlo_integral
```

## Question 5 Part A

Make use of dictionary to create a character translator function *translate*. It takes 3 strings as arguments: *translate(source, destination, string)*. *source* contains the set of characters you want "translated", *destination* contains the set of characters to translate to, and *string* is the string to perform the translation on.

## Question 5 Part A Discussion

Recall the usage of dictionary

- You should be storing *source* as the keys
- And *destination* as the values
- What should you use to check key-value pair?
- Make sure no key error will happen  $=p$



## Question 5 Part A Zexin's solution

```
def translate(source, destination, string):  
    l = len(source)  
    d = dict()  
    for i in range(l):  
        d[source[i]] = destination[i]  
    newString = ""  
    for char in string:  
        if char in d:  
            newString += d[char]  
        else:  
            newString += char  
    return newString
```

## Question 5 Part B

Create a function *caesar\_cipher(shift, string)*, where *shift* is the number of positions to shift, and *string* is the string to encrypt.

## Question 5 Part B Discussion

You are not restricted to only use Dictionary now

- You are instead encouraged to use *chr* and *ord*
- Take note of what is ASCII code
- Take note how to check uppercase or lowercase

## Question 5 Part B Zexin's solution

```
def caesar_cipher(shift, string):  
    newString = ""  
    for char in string:  
        if char.isupper():  
            original = 'A'  
        else:  
            original = 'a'  
        new = chr((ord(char) - ord(original) + shift) % 26 + ord(original))  
        newString += new  
    return newString
```

## Extra stuff: A simple final question on Dictionary

If time permits, we will go through this.

```
s = "cs1010s c001" # last character is digit one
d = {}
for i in range(len(s)):
    d[s[i]] = i
print(d)
```

- This is a problem for which you need to know basics of Dictionary.

## Extra stuff: A simple final question on Dictionary

```
s = "cs1010s c001" # last character is digit one
d = {}
for i in range(len(s)):
    d[s[i]] = i
print(d)
```

- $i$  will loop from 0 to index of last character
- Each time, the value corresponding to character is updated to  $i$
- What will be the result?

## Extra stuff: A simple final question on Dictionary

```
s = "cs1010s c001" # last character is digit one
d = {}
for i in range(len(s)):
    d[s[i]] = i
print(d)
```

Hence the solution is:

{ 'c': 8, 's': 6, '1': 11, '0': 10, ' ': 7 }

Note that the key-value pairs in your answer can be in any order.

- Slides + relevant material available at:

`https://github.com/wangzexin/Teaching`

- After the tutorial, if you have further questions:

`wang.zexin@u.nus.edu`



# Thank You

*wang.zexin@u.nus.edu*