National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2024/2025

Tutorial 8
Message Passing & Stateful Objects

1. The following code defines a new *widget* object in *message passing* style:

```
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

widget = make_widget()
```

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

   (b) Write a program to insert the following objects into the *widget* object named `widget`: 1, 2, 3.

   (c) Suppose we perform a retrieval 3 times. What is returned each time?

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.

   Sample execution:

```
>>> A = make_accumulator()
>>> A(10)
10
>>> A(10)
20

>>> B = make_accumulator()
>>> B(30)
30
>>> B(-10)
20
```

3. In software-testing applications, it is useful to be able to count the number of times a given function is called during the course of a computation. 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.

(a) Implement make_monitored. For instance, we could make a monitored version of the sqrt function:

```
>>> s = make_monitored(sqrt)
>>> s(100)
10.0
>>> s("how-many-calls?")
1
>>> s(1024)
32.0
>>> s("how-many-calls?")
2
>>> s("reset-count")
>>> s("how-many-calls?")
0
```

(b) Explain how you can extend make_monitored so that it works for functions that take an arbitrary number of parameters. Rewrite your function such that it implements your proposal. For your information:

```
def sum_numbers(*numbers):
    s = 0
    for n in numbers:
        s += n
    return s

>>> sum_numbers(1,2,3)
6
>>> sum_numbers(1,2,3,4,5)
15
```

4. *Monte Carlo integration* is a method of estimating definite integrals by means of Monte Carlo simulation.

Consider computing the area of a region of space described by a predicate $P(x, y)$ that is true for points (x, y) in the region and false for points not in the region. For example, the region contained within a circle of radius 3 centered at $(5, 7)$ is described by the predicate that tests whether $(x - 5)^2 + (y - 7)^2 < 3^2$.

To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at $(2, 4)$ and $(8, 10)$ contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region.

We can estimate the integral by picking, at random, points (x, y) that lie in the rectangle, and testing $P(x, y)$ for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

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.

This new function accepts the following commands:

(a) "run trials" and a number of trials to run and performs the stated number of trial.

(b) "trials" will return the number of trials run thus far.

(c) "get estimate" to return the current estimate given the trials run so far.

Use your Monte Carlo integral to produce an estimate of the area of a unit circle. *Hint:* To choose a number at random from a given range, you can use `random.uniform(low, high)` after `import random`

Sample execution (Note: your values for `circle_estimate` should be roughly approximate but probably will not be exactly the same):

```
>>> circle_estimate =  make_monte_carlo_integral(circle,-1,-1,1,1)
>>> circle_estimate("run trials", 1000)
>>> print(circle_estimate("trials"))
1000

>>> print(circle_estimate("get estimate"))
3.068

>>> circle_estimate("run trials", 10000)
>>> print(circle_estimate("trials"))
11000

>>> print(circle_estimate("get estimate"))
3.1243636363636362

>>> circle_estimate("run trials", 100000)
>>> print(circle_estimate("trials"))
111000

>>> print(circle_estimate("get estimate"))
3.1454774774774776
```

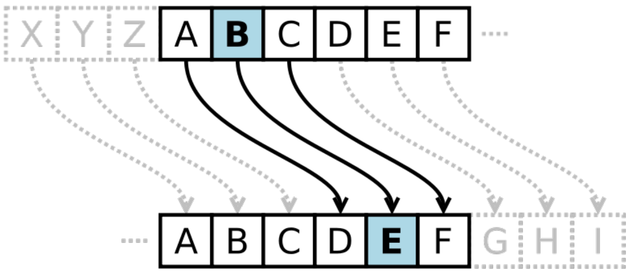
5. Translations

(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. For example:

```
>>> translate("dikn","lvei","My tutor IS kind")
"My tutor IS evil"
```

In the example we have: $d \to l, i \to v, k \to e, n \to i$, thus `kind` is translated to `evil`.

(b) A *Caesar cipher* is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would replace A, E would replace B, and so on. As illustrated in the following graph:



Create a function `caesar_cipher(shift, string)` , where `shift` is the number of positions to shift, and `string` is the string to encrypt.
(*Hint:* use `translate` in your implementation).

```
>>> caesar_cipher(29,"aAbB")
"dDeE"
```

(Alternatively, the `ord()` and `chr()` could be useful here. You can read more here: http://docs.python.org/3.6/library/functions.html.)