Day-4

Functional Programming

Programming paradigm

- **Programming paradigm**
  - 절차적(procedural) (구조적) 프로그래밍 : C, pascal
    - ✓ 코드(함수)중심의 데이터, 알고리즘과 로직에 의거하여 단계를 밟아가며 문제를 해결하는 프로그래밍 언어
  - 객체지향형 (object oriented) 프로그래밍 : java
    - ✓ 데이터중심의 코드, 객체를 만들고 객체를 조립하는 것을 목표로 한 언어
  - 함수형 (functional) 프로그래밍 : Haskell
    - ✓ 함수를 데이터처럼 취급이 가능

- **Python 은?**
  - 절차지향적, 객체지향형, 함수형 지원 모두 지원 (다중 패러다임)

- **파이썬은 어떤 프로그래밍 패러다임을 이용해야 효율적일까?**
  - 모두 다 괜찮다. (적재적소)

Functional programming

- **프로그래밍 패러다임 중 하나**

- **Functional programming의 특징**
  - **① 함수=1st class(object)**: 함수를 데이터처럼 취급 가능
  - **② comprehension**': 어떻게'보다는 '무엇'을 계산하는지가 중요
  - **③ map, reduce 등** : 주요 제어 구조로 recursion 활용(loop X)
  - **④ high order 함수**(함수에 대해 연산하는 함수) 지원

Functional programming

- Python의 functional programming 특징
  - ① lambda와 map / filter / reduce
  - ② iterable / iterator / generator
  - ③ 함수는 객체이다.
  - ④ high order function과 decorator / closure

lambda 함수

함수

- 프로그램에서 자주 사용되는 코드를 따로 만들어 두고 필요할 때마다 불러서(호출해서) 사용하는 기능

• 일반적인 함수 예1

```
def click(x):
    print (x+5)
button_text = 3
click(button_text)
```

• 일반적인 함수 예2

```
button_text = 3
def click(x=button_text):
    print (x+5)
click( )
```

이름없는 함수 lambda

- lambda 함수
  - 함수 이름을 만들지않고 함수 기능은 수행하도록 하는 기능

• lambda 함수 정의 형식

```
lambda  매개변수1, ..., 매개변수n : 표현식
```

• 이름없는 함수 정의 예

```
button_text = 3
lambda  x=button_text : print (x+5)
```

함수이름부분이 없음

함수실행문장

이름없는 함수 lambda

- lambda 함수

함수 이름이 없는데 어떻게 호출하지요?

lambda함수를 호출할 때는 변수를 마치 함수처럼 활용합니다.

- 변수를 함수처럼 호출하는 예

```
button_text = 3
var = lambda  x=button_text : print (x+5)
var( )
```

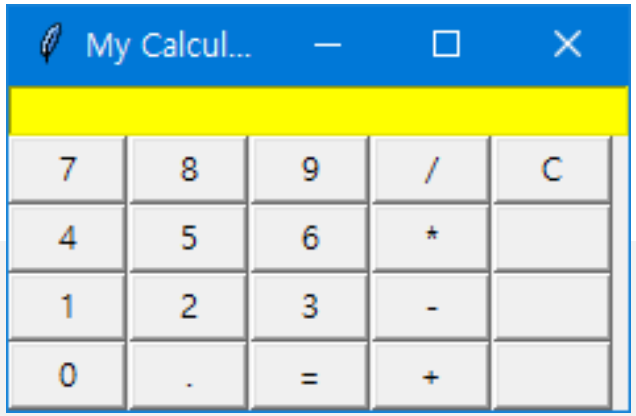
이름없는 함수 lambda

- **lambda 함수의 장점**
  - 메모리 절약 + 가독성 향상
    - ✓ 일반 함수는 객체를 만들고, 재사용을 위해 함수 이름(메모리)을 할당한다.
    - ✓ lambda 함수는 한번 사용하고 다음 줄로 넘어가면 메모리(힙(heap) 영역)에서 사라짐
      - ❖ (참고) 가비지 컬렉터(참조하는 객체가 없으면 지워진다)
      - ❖ 파이썬은 모든 것이 객체로 관리, 객체들은 레퍼런스 카운터 존재. 이 카운터가 0이 되면, 즉 누구도 참조하지 않으면 메모리를 환원

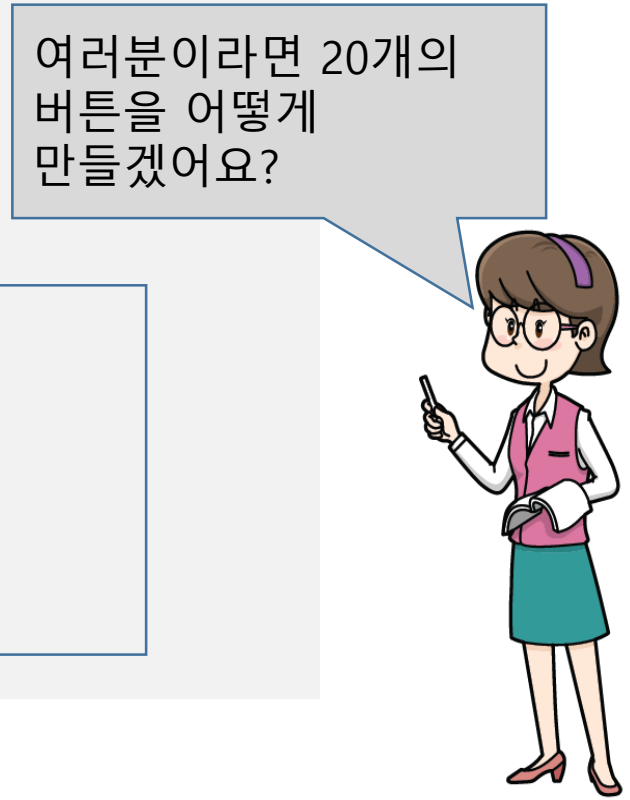
계산기 프로그램에서 lambda함수 활용해보기

실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기




```
from tkinter import *

window = Tk()
window.title("My Calculator")
display = Entry(window, width=33, bg="yellow")
display.grid(row=0, column=0, columnspan=5)
```

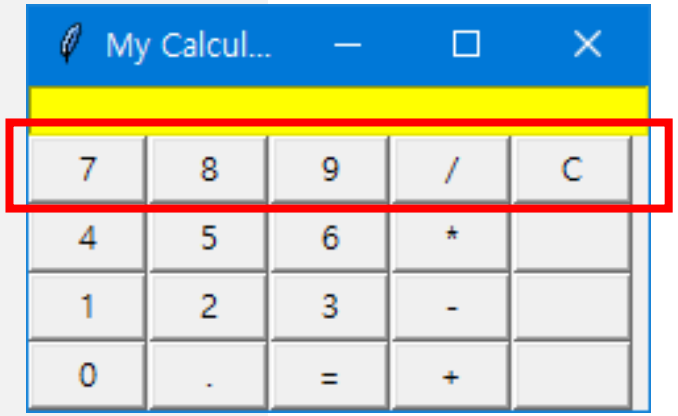


실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기

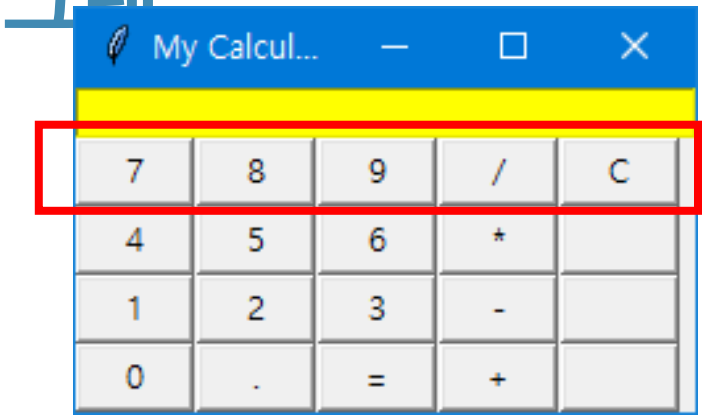
```
from tkinter import *

window = Tk()
window.title("My Calculator")
display = Entry(window, width=33, bg="yellow")
display.grid(row=0, column=0, columnspan=5)

button_list =[
   '7', '8', '9', '/', 'C' ]
```

먼저, 버튼 5개만 만들어 볼까요?
리스트를 활용해 보기로 하죠

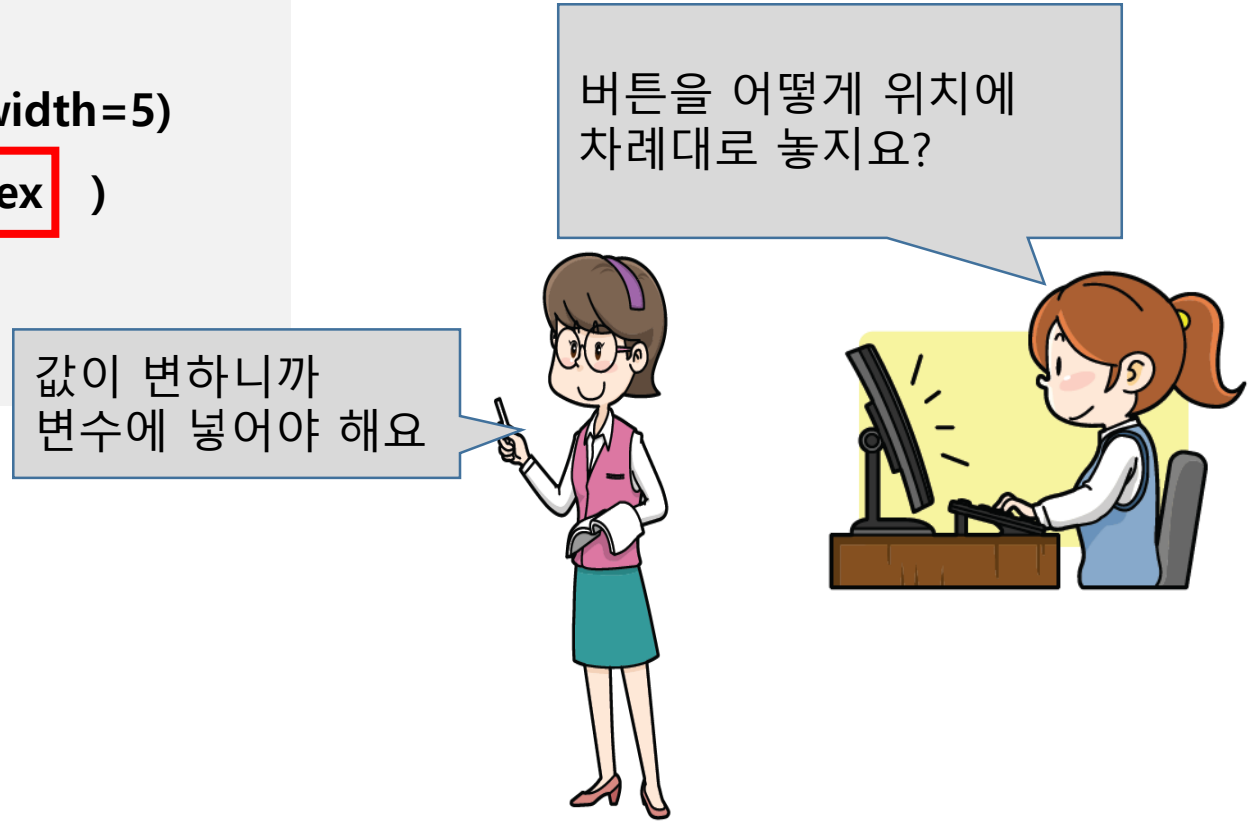


실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기

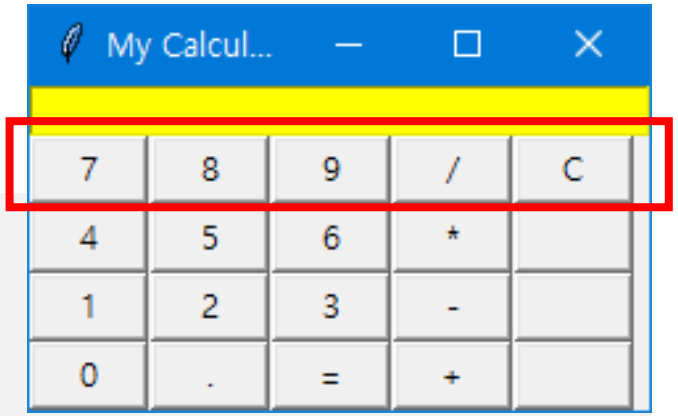



```
.....
for button_text in button_list:
    btn = Button(window, text=button_text, width=5)
    btn.grid(row= 1 , column= col_index  )

window.mainloop()
```



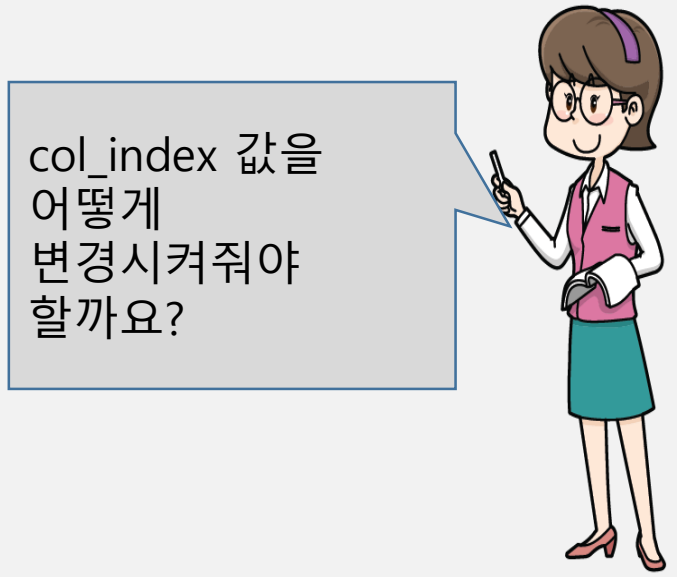
실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기



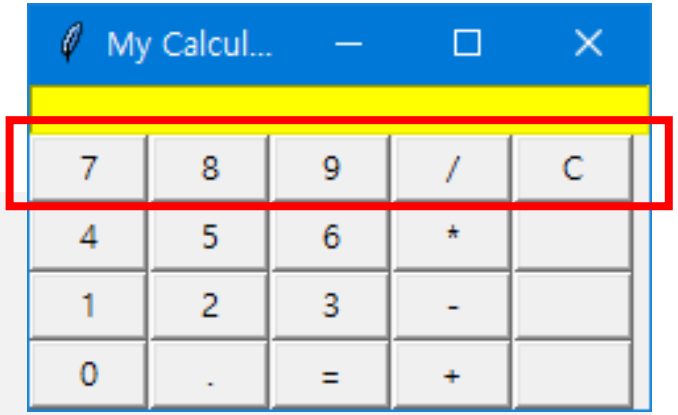
col_index의 초기값

```
col_index = 0
for button_text in button_list:
    btn = Button(window, text=button_text, width=5)
    btn.grid(row=1, column=col_index )

window.mainloop()
```



실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기

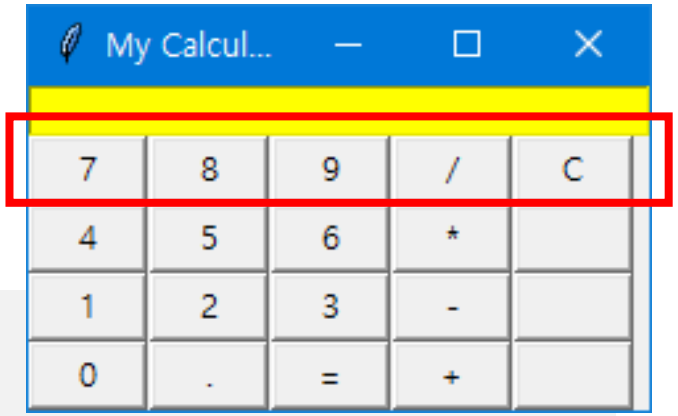



```
col_index = 0
for button_text in button_list:
    btn=Button(window, text=button_text, width=5)
    btn.grid(row=1 , column=col_index)
    col_index += 1    # col_index = col_index + 1과 같은 의미

window.mainloop()
```

col_index를
변경시켜준다.

실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기



버튼에 대한 변수가 필요한 경우가 아니라면 꼭 변수를 만들지않아도 됨
이와 같이 한 줄의 코드로 버튼을 만들면서 배치시킬 수 있음

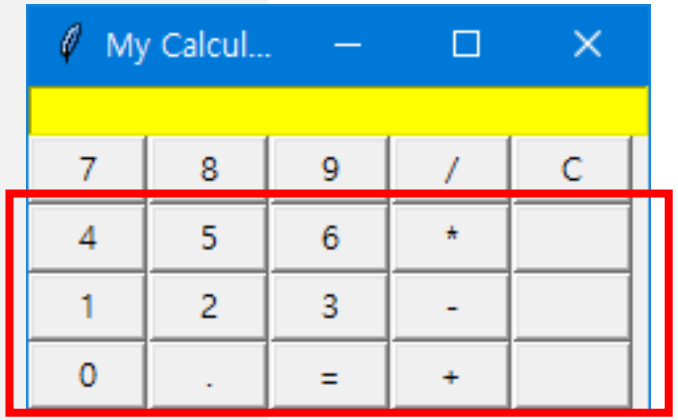
```
row_index = 1
col_index = 0
for button_text in button_list:
    Button(window, text=button_text, width=5).grid(row=1, column=col_index)
    col_index += 1    # col_index = col_index + 1과 같은 의미


window.mainloop()
```

실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기

```
from tkinter import *

window = Tk()
window.title("My Calculator")
display = Entry(window, width=33, bg="yellow")
display.grid(row=0, column=0, columnspan=10)

button_list = [
'7', '8', '9', '/', 'C',
'4', '5', '6', '*', ' ',
'1', '2', '3', '-', ' ',
'0', '.', '=', '+', ' ' ]
```

나머지 버튼도 만들어보죠
리스트의 아이템 추가

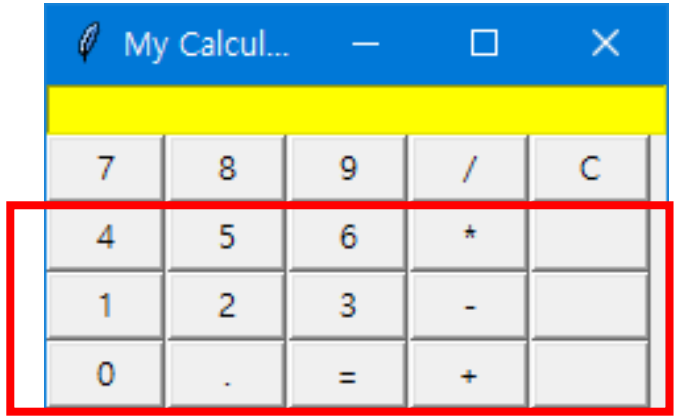


실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기

```
.....
for button_text in button_list:
    btn = Button(window, text=button_text, width=5)
    btn.grid(row= row_index , column= col_index   )
    col_index += 1
    ...
window.mainloop()
```

1에서 row_index로 변경

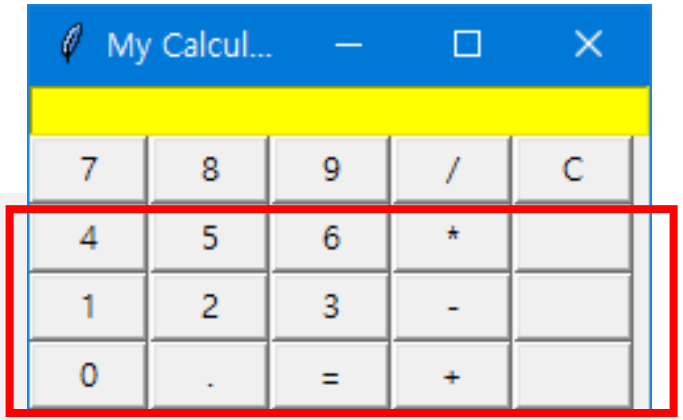


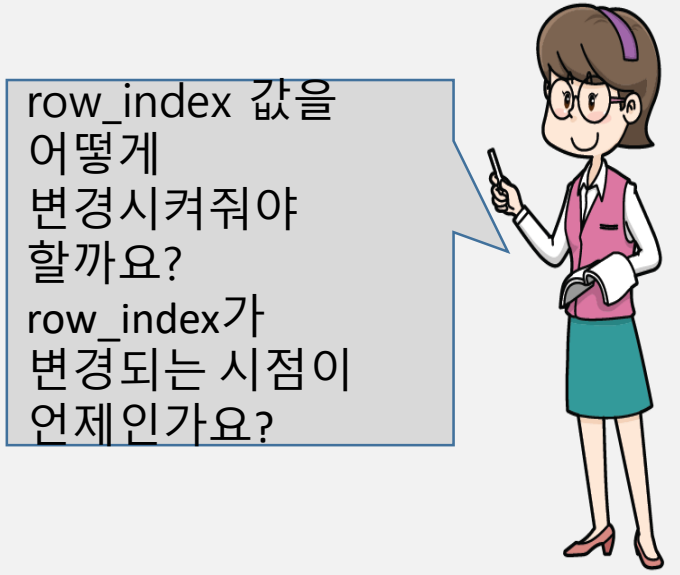
실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기

```
row_index = 1
col_index = 0
for button_text in button_list:
    btn = Button(window, text=button_text, width=5)
    btn.grid(row=row_index, column=col_index )
    col_index += 1
    ....


window.mainloop()
```

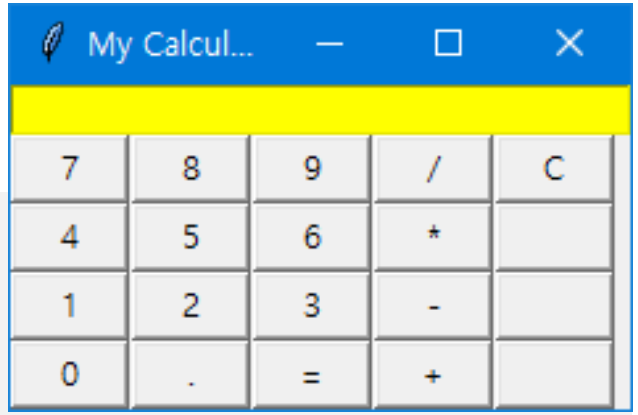
row_index의 초기값 추가





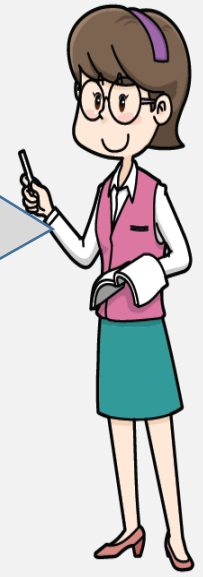
실습 : lambda를 활용한 계산기 프로그램

- 1단계 : 화면구성하기



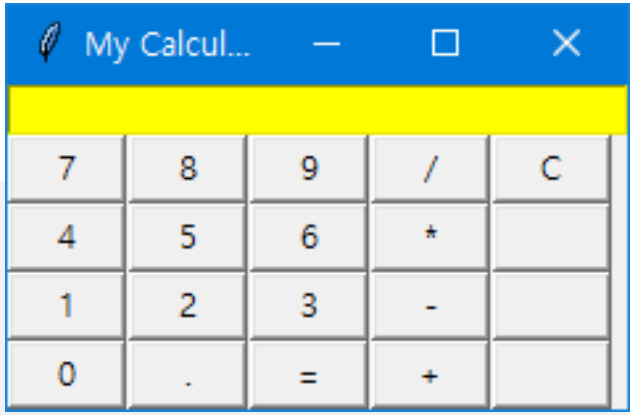

```
row_index = 1
col_index = 0
for button_text in button_list:
    btn=Button(window, text=button_text, width=5)
    btn.grid(row=row_index, column=col_index)
    col_index += 1    # col_index = col_index + 1과 같은 의미
    if col_index > 4:
        row_index += 1
        col_index = 0

window.mainloop()
```

row_index가 증가하는 시점은 col_index가 4보다 클때입니다. 그리고 그 시점은 다시 col_index가 0으로 되는 시점이기도 합니다.



실습 : lambda를 활용한 계산기 프로그램

- 2단계 : 버튼 이벤트 연동하기




```
def click(key):
    display.insert(END, key)
```

엔트리박스에 클릭한 버튼의 값을 나타낸다.

```
row_index = 1
col_index = 0
for button_text in button_list:
    Button(window, text=button_text, width=5, command=click(button_text))
                .grid(row=row_index, column=col_index)
    col_index += 1
    if col_index > 4:
        row_index += 1
        col_index = 0
```

길어서 줄을 바꾼 것임
여러분은 한 줄로 쓰세요

command 옵션에 함수 연동 시, ( )를 넣으면 버튼을 만들면서 바로 함수 호출이 되어 실행이 됩니다.

실습 : lambda를 활용한 계산기 프로그램

- lambda 함수를 이용

계산기 프로그램에서는 숫자 버튼을 누르면 함수를 호출하는 것이므로 호출했을 때 실행할 진짜 함수를 실행 문장에 넣어주면 됩니다.

```
def click(key):
    display.insert(END, key)


lambda  x=button_text : click(x)
```

함수실행문장

실습 : lambda를 활용한 계산기 프로그램

- 2단계 : 버튼 이벤트 연동하기

```
def click(key):
    display.insert(END, key)


row_index = 1
col_index = 0
for button_text in button_list:
    Button(window, text=button_text, width=5, command= lambda x = button_text : click(x))
                    .grid(row=row_index, column=col_index)
    col_index += 1
    if col_index > 4:
        row_index += 1
        col_index = 0
```

실습 : lambda를 활용한 계산기 프로그램

- 3단계 : 계산기 버튼 기능 넣기

```
def click(key):
    if key == '=':
        result = eval(display.get())
        s = str(result)
        display.insert(END, '=' + s)
    else:
        display.insert(END, key)
```

'=' 기호를 클릭했을 때는 계산을 해주어야 함

eval() 함수: 수식을 받아서 계산을 수행하는 함수

- eval()함수

```
>>> eval(3+5)
TypeError: eval() arg 1 must be a string
>>> eval('3+5')
8
```

eval() 함수는 문자로 된 수식만 계산을 합니다.

실습 : lambda를 활용한 계산기 프로그램

- **난이도:중하**
  - "%" 를 계산기 프로그램 버튼의 빈 공간에 추가해서 나머지를 구할 수 있게 만들어보세요.
  - "**" 를 계산기 프로그램 버튼의 빈 공간에 추가해서 제곱값을 구할 수 있게 만들어보세요.
  - "C" 클릭하면 엔트리박스에 값들이 모두 지워지게 해보세요.

- **난이도:상**
  - " ←"를 계산기 프로그램 버튼의 빈 공간에 추가해서 한글자씩 지우게 만들어보세요.
    - ✓ 힌트 : len()함수를 이용하면 됩니다.

High order function

High order function

- **High order function(고계함수 or 고차함수)**
  - 람다나 다른 함수를 인자로 받거나 함수를 반환하는 함수
  - 고계함수는 lambda()함수와 함께 주로 사용됨
  - map(), reduce(), filter()
  - 맵리듀스 알고리즘(map-reduce 알고리즘) : 대용량 분산처리시스템에서 각각의 work을 쪼개서 여러시스템에 나누어주고 (map) 다시 그 결과를 합쳐서 (reduce) 반환해주는 알고리즘

map 함수

map 함수

- 함수명과 반복에 사용될 아이템을 전달인자로 받아 함수에 의해 수행된 결과를 (분배하여) 반환해주는 함수

• map() 형식

(map(함수명, 반복에 사용될 아이템))

- list()와 함께 사용
- 아이템에는 순서형자료가 들어감
  - ✓ list, dictionary, set, tuple, 문자열 등

• map() 예

```
>>>def wow(text):
                return text.upper()

>>>wow('hi')
HI

>>>list(map(wow, ['hi', 'hello', 'welcome']))
['HI', 'HELLO', 'WELCOME']

>>>map(wow, ['hi', 'hello', 'welcome'])
<map object at 0x034069F0>
```

map 함수

```
result = map(lambda x: x/10, range(10))
for x in result:
    print(x)

result = list(map(lambda x: x/10, range(10)))
```

map 함수

- **map 함수가 없다면?**
  - 1~10, 1~100, 1~1000까지의 합을 출력하는 프로그램을 함수로 작성하시오.

```
>>>
1부터~10까지의 합:55
1부터~100까지의 합:5050
1부터~1000까지의 합:500500
```

map 함수

- map함수를 사용하면?
  - 1~10, 1~100, 1~1000까지의 합을 출력하는 프로그램을 함수로 작성하시오.

```
def summation(value):
    num = 1
    total = 0
    while   num <= value:
        total = total + num
        num = num  + 1
    print('1부터 '+ str(value) +'까지의 합: '+ str(total))
list(map(summation, [10, 100, 1000]))
```

```
>>>
1부터~10까지의 합:55
1부터~100까지의 합:5050
1부터~1000까지의 합:500500
```

실습

- map 과 lambda 함수를 사용하여 리스트 [1, 2, 3]을 주어 제곱의 값을 리스트로 출력해 주는 코드를 작성하시오.

In :	
Out:	**[1, 4, 9]**

reduce 함수

reduce 함수

- **함수명와 반복에 사용될 아이템을 전달인자로 받아 함수에 의해 수행된 결과를 (줄여서) 리스트로 반환해주는 함수**

• reduce() 형식

```
from functools import reduce
reduce(함수명, 반복에 사용될 아이템)
```

- 반드시 함수에 두개의 전달 인자가 있어야 함
- 아이템에는 순서형자료가 들어감
  - ✓ list, dictionary, set, tuple, 문자열 등

• reduce() 예

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [0,1,2,3,4])
10
```

0과 1을 더하고, 그 결과에 2를 더하고.... 더하는 과정을 반복 ((((1+2)+3)+4)+5)

실습

- reduce함수를 사용하여 'abcd'를 역순으로 출력하는 코드를 작성하시오.

In :	
Out:	'dcba'

filter 함수

filter 함수

- 함수명과 반복에 사용될 아이템을 전달인자로 받아 함수에 의해 수행된 결과를 (필터하여) 리스트로 반환해주는 함수

- filter() 형식

```
list(filter(함수명, 반복에 사용될 아이템))
```

  - list()함수와 함께 사용
  - 아이템에는 순서형자료가 들어감
    - ✓ list, dictionary, set, tuple, 문자열 등

- filter() 예

```
>>> list(filter(lambda x: x<5, range(10)))
[0,1,2,3,4]

>>> list(filter(lambda x: x>5, range(10)))
[6,7,8,9,10]

>>> filter(lambda x: x>5, range(10))
<filter object at 0x03393F50>
```

실습

- filter함수를 사용하여 0~9사이의 숫자 가운데 홀수만 출력하는 코드를 작성하시오.

In :	
Out:	[1,3,5,7,9]

iterator

iterator 객체

- **iterator 객체**
  - map(), filter() 함수 등을 이용해서 만든 객체
  - iterator 객체와 관련된 메소드
    - ✓ next() : iterator 객체의 아이템들을 순차적으로 호출하는 메소드
    - ✓ iter() : 반복 가능한 데이터를 입력 받아 iterator 객체로 만드는 메소드

iterator 객체

- next() : iterator 객체의 아이템들을 순차적으로 호출하는 메소드

```
>>> lst = [1,2,3]
>>> result = map(lambda i: i**2, lst)
>>> next (result) #next()에 의해 iterator 아이템을 반환하고 실행을 멈춤
1
>>> next (result) #next() 에 의해 다시 호출되면 멈춤 위치에서부터 다시 실행
4
>>> next (result)
9
>>> result = filter(lambda x: x>5, range(10))
>>> next (result)
6
>>> next (result)
7
```

iterator 객체

- iter() : 반복 가능한 데이터를 입력 받아 iterator 객체로 만드는 메소드

```
>>> a=iter('123')  #a는 iterator 객체임
>>> next (a)
'1'
>>> next (a)
'2'
>>> next (a)
'3'
>>> b= list('123')
>>> next (b)
TypeError: 'list' object is not an iterator
>>> c=iter([1,2,3])
>>> next (c)
1
>>> next (c)
2
>>> d=[1,2,3]
>>> next (d)
TypeError: 'list' object is not an iterator
```

generator

generator 객체

- **사용자 정의 함수를 통해 iterator 가능한 아이템을 생성하여 가지고 있는 객체**
  - 사용자 정의 함수 안에 yield를 사용하여 generator 객체의 아이템을 생성
  - 사용자 정의 함수를 호출하여 generator 객체를 만듦

  - generator 함수 정의와 generator 객체 생성

```
>>> def abc():   #generator 객체 정의
        yield 'a'   #generator 객체의 아이템 생성
        yield 'b'
        yield 'c'
>>> abc()  #generator 객체 생성
<generator object abc at 0x033ED240>
>>> g_obj=abc() #generator 객체를 생성
>>> next (g_obj)
'a'
>>> next (g_obj)
b'
```

generator 객체

- generator의 next()에 의해 yield 문을 만나면 iterator 아이템을 반환하고 실행을 멈춤
- next() 에 의해 다시 호출되면 멈춤 위치에서부터 다시 실행

- generator 함수 정의

```
>>> def abc():
        yield 'a'
        yield 'b'
        yield 'c'
>>> a=abc()
>>> print (a)
<generator object abc at 0x02FFE0C0>
>>> next(a)
a
```

- 일반 함수 정의

```
>>> def efg():     #함수정의
        print ('hello')
>>> b=efg()
>>> print (b)
None
```

generator 객체

```
>>> def efg():
        print ('hello')
        yield 'a'
        yield 'b'
>>> b=efg()
>>> print (b)
<generator object abc at 0x02FFE0C0>
>>> next(b)
hello
'a'
>>> next(b)
hello
'b'
```

generator 객체

- **generator는 어디에 활용할까?**
  - 컴퓨터 메모리는 한정되어 있으므로 순회해야 할 데이터를 미리 정의해 두는 것은 불가능
    - ✓ 예. 1~무한대까지 모든 자연수를 담은 리스트를 컴퓨터에 미리 만들어 저장할 수 없다.
  - generator를 통해 필요한 값을 하나씩 꺼내도록 하면 무한대 리스트를 흉내 낼 수 있다.

```
>>> def one_to_infinite (): #무한대의 자연수 순서대로생성하기
        num = 1
        while True:
            yield n
            num = num  + 1
>>> natural_number = one_to_infinite()
>>> next(natural_number)
1
>>> next(natural_number)
2
```

실습

- generator 를 사용하여 리스트와 튜플로 iterable한 데이터 만들기

```
def countdown (start, end):
    

print (list(countdown(10,0)))
print (tuple(countdown(20,10)))
```

실습

- 0 ~ 45 랜덤한 값을 리스트로 만드시오.

```
import random
randomlist= [ ]
for x in range(5):
    randomlist.append(0,15)

print (randomlist)
```

generator expression

generator expression

- generator expression

  - generator expression 형식

```
연산 for 변수 in 컬렉션
```

  - 예제)리스트 조건제시법으로 0~45사이 랜덤한 값 5개 만들기

```
[random.randint(0,45) for x in range(5)]
```

  - 예제)리스트 조건제시법으로 세제곱수 10개 만들기

generator expression

- generator expression으로 만든 generator 역시 next() 함수로 다음 아이템을 갖고 올 수 있다.

```
>>> x_generator = (x ** 3 for x in range(10))

>>> next(x_generator)
0
>>> next(x_generator)
1
>>> next(x_generator)
8
```

실습 예제

- **주문받기**
  - input_order() 함수는 사용자로부터 n 개의 주문을 받아 리스트로 반환
  - 이 함수를 이용해 음료를 3개 입력 받고 제조를 지시

```
def input_order(n):
    ''' n 개의 음료를 주문받아 리스트로 반환하는 함수'''
     return [input() for x in range(n)]

#음료주문 3개 입력 받아 각 음료마다 제조 지시
for drink in input_order(3):
    print (drink, '만들어주세요')
```

입력값:

아메리카노
카페라떼
딸기쥬스

출력값:
아메리카노 만들어주세요.
카페라떼 만들어주세요.
딸기쥬스 만들어주세요.

실습 예제 솔루션

```
def input_order(n):
    ''' n 개의 음료를 주문받아 리스트로 반환하는 함수'''
     return (input() for x in range(n))

#음료주문 3개 입력 받아 각 음료마다 제조 지시
for drink in input_order(3):
    print (drink, '만들어주세요')
```

입력값:

아메리카노
아메리카노 만들어주세요.
카페라떼
카페라떼 만들어주세요.
딸기쥬스
딸기쥬스 만들어주세요.

함수는 객체다

함수는 객체다

- **함수를 데이터처럼 취급 가능**
  - 예제1: 함수를 일반 데이터처럼 전달 가능

```
In [ ]:  def add(x, y):
             'Performs x + y'  # 첫 줄의 문자열은 도움말로 취급됨
             return x + y
In [ ]:  newAdd = add          # 함수를 데이터처럼 변수에 대입
         del add               # add함수 삭제
         newAdd(3, 4)          # newAdd함수를 여전히 사용 가능
In [ ]:  newAdd.__doc__        #__doc__: 도움말
In [ ]:  help(newAdd)
In [ ]:  newAdd.__name__       #__name__: 정의되었던 함수 이름
```

  - 모든 함수는 생성되는 시점에 구분자(identifier)가 붙음 (디버깅을 지원하기 위한 목적)
    - ✓ 예) _name_, _doc_

함수는 객체다

In :	def wow(text): return text.upper()
In :	wow('hi')
Out:	'HI'
In :	oh = wow
In :	oh('hello')
Out:	'HELLO'
In :	del wow
In :	oh('hello~')
Out:	'HELLO~'
In :	wow('hello?')
Out:	NameError: name 'wow' is not defined

- 참고
  - ✓ upper() : 대문자로 변환
  - ✓ lower() : 소문자로 변환
  - ✓ capitalize(): 첫 글자만 대문자로 변환

함수는 객체다

- 예제2: 함수를 자료구조 속에 아이템으로 넣을 수 있음

In :	flst = [oh, str.lower, str.capitalize]

In :	flst
Out:	[<function __main__.wow(text)>, <method 'lower' of 'str' objects>, <method 'capitalize' of 'str' objects>]

In :	for x in flst: print(x('welcome'))
Out:	WELCOME welcome Welcome

oh('welcome')
str.lower('welcome')
str.capitalize('welcome')

In :	flst[0]('welcome~')
Out:	'WELCOME~'

객체를 꺼내서 변수에 넣지 않고도 리스트 안에 있는 함수를 호출 할 수 있음

함수는 객체다

In [ ]:
```
def yell(text):
    return text.upper() + '!!!'
def whisper(text):
    return text.lower() + '..'

funcs = [str.capitalize, yell, whisper]

for f in funcs:
    print(f('get out'))
```

In [ ]:
```
funcs[0]('be quiet')
```

In [ ]:
```
funcs[1]('Be quiet')
```

In [ ]:
```
funcs[2]('Be quiet')
```

함수는 객체다

- 예제3: 함수를 함수의 인자로 전달 가능

In [ ] :
```
def cal(f, x, y):
    return f(x, y)

def add(x, y):
    return x + y
def sub(x, y):
    return x - y
```

In [ ] :
```
cal(add, 3, 4)
```

In [ ] :
```
cal(sub, 3, 4)
```

함수는 객체다

In :	def greet(func):
	    greeting = func('hi, I love python')
	    print(greeting)

In :	greet(oh)
Out:	HI, I LOVE PYTHON

내부함수

내부 함수

- **함수 내 함수 (inner/nested function)**
  - Python은 함수 내에 다른 함수가 정의되는 것을 허용
  - 이런 함수를 중첩 함수(nested function) 또는 내부 함수(inner function)라 함
  - 정의된 함수 내부에서만 사용 가능

내부 함수

- **함수 내 함수**
  - 예)함수 speak를 실행할 때마다 함수 wow를 만들고 호출

```
In :  def speak (text):
          def wow(t):
              return t.lower()
          return wow(text)
```

```
In :  speak('Hello, World')
Out:  'hello, world'
```

내부 함수

- **함수 내 함수**
  - 예) 함수 speak밖에서는 함수 wow의 존재를 모름

In :	**wow('Hi')**
Out:	**NameError**: name 'wow' is not defined

In :	**speak.wow**
Out:	**AttributeError**: 'function' object has no attribute 'wow'

내부 함수

- 내부함수 예1)

In [ ] :
```
def speak(text):
    def yell(t):
        return t.upper() + '!!!'

    return yell(text)
```

In [ ] :
```
speak('hello')
```

In [ ] :
```
yell('hello')
```

In [ ] :
```
speak.yell('hello')
```

내부 함수

- 내부함수 예2)

In :	def get_speak_wow(volume):
	    def wow(text):
	        return text.lower()
	    def oh(text):
	        return text.upper()
	    if volume > 0.5:
	        return oh
	    else:
	        return wow

In :	speak_func = get_speak_wow(0.8) speak_func('Hello')
Out:	'HELLO'

내부 함수

- 내부 함수 예3)

In [ ] :
```
def getSpeak(volume):
    def yell(text): return text.upper() + '!!!'
    def whisper(text): return text.lower() + '..'

    if volume >= 5: return yell
    else: return whisper
```

In [ ] :
```
speak = getSpeak(7)
speak('hello')
```

In [ ] :
```
getSpeak(7)('hello')
```

In [ ] :
```
speak = getSpeak(2)
speak('hello')
```

내부 함수

- 내부 함수 예4)

In [ ] :
```
def getCal(op):
    def add(x, y): return x + y
    def sub(x, y): return x - y
    def mul(x, y): return x * y
    def div(x, y): return x / y

    return {'+': add, '-': sub, '*':mul, '/':div }.get(op)
```

참고 : 딕셔너리 에서 .get(key) 함수 : key로 value 얻기

In [ ] :
```
cal = getCal('+')
cal(3, 4)
```

In [ ] :
```
cal = getCal('*')
cal(3, 4)
```

클로저

클로저(closures)

- 내부 함수는 상위 함수의 데이터 접근이 가능함
  - 예1)

```
In :  def get_speak_func(text, volume):
          def wow():
              return text.lower()
          def oh():
              return text.upper()
          if volume > 0.5:
              return oh
          else:
              return wow
```

함수 get_speack_func()는 text 와 volume 두개의 전달인자를 가짐

wow, oh 내부 함수는 전달인자를 가지고 있지 않음

그러나 상위함수 get_speak_func()에서 받은 text 변수에 접근이 가능

```
In :  get_speak_func('Hello, World', 0.7)()
Out:  'HELLO, WORLD'
```

클로저(closures)

- 예2)

In [ ]:
```
def getSpeak(text, volume):
    def yell(): return text.upper() + '!!!'
    def whisper(): return text.lower() + '..'

    if volume >= 5: return yell
    else: return whisper
```

In [ ]:
```
getSpeak('hello', 7)()
```

In [ ]:
```
getSpeak('hello', 2)()
```

클로저(closures)

- 예3)

In [ ] :
```
def cal(x, y, op):
    def add(): return x + y
    def sub(): return x - y
    def mul(): return x * y
    def div(): return x / y

    return {'+': add, '-': sub, '*':mul, '/':div }.get(op)()
```

In [ ] :
```
cal(7, 2, '+')
```

In [ ] :
```
cal(7, 2, '-')
```

In [ ] :
```
cal(7, 2, '*')
```

In [ ] :
```
cal(7, 2, '/')
```

데코레이터

decorator

- 꾸미다(decorate) + er(or) = 장식하는 도구
- 함수(메서드)를 장식할때 사용하는 함수
- 함수의 수정없이 함수의 앞뒤로 기능을 붙일때 사용
- @을 사용하여 쉽게 표현이 가능

- decorator 정의와 호출

```
def decorator(func) :
    def wrapper():
        statements0
        func()
        statements1
    return wrapper
```

호출

```
decorated_func = decorator(func)
decorated_func()
```

or

```
@decorator
func()
```

decorator

- 데코레이터를 사용하지 않은 일반 함수 작성의 예

```
def hello() :
    print('함수시작')
    print('hello')
    print('함수끝')

def world() :
    print('함수시작')
    print('world')
    print('함수끝')
```

```
hello()
함수시작
hello
함수끝

world()
함수시작
world
함수끝
```

decorator

- 데코레이터를 사용하지 않고 내부함수로 작성한 예)

```
def trace(func) :
    def wrapper():
        print('함수시작')
        func()
        print('함수끝')
return wrapper
```

```
def hello() :
        print('hello')
def world() :
        print('world')
```

```
trace_hello = trace(hello)
trace_hello()

함수시작
hello
함수 끝
```

```
trace_world = trace(world)
trace_world()

함수시작
world
함수 끝
```

decorator

- @ 엣을 이용한 데코레이터를 사용한 예

```
def trace(func) :
    def wrapper():
        print('함수시작')
        func()
        print('함수끝')
return wrapper
```

```
@trace
def hello() :
        print('hello')

@trace
def world() :
        print('world')
```

```
hello()
함수시작
hello
함수 끝
world()
함수시작
world
함수 끝
```

Thank you!

Beyond The Engine of Korea

HANYANG UNIVERSITY

한양대학교
HANYANG UNIVERSITY