

Closure

순서

- 클로저 발표배경
 - 보통 개발자들의 인식
- 들어가기전에 앞서서
 - 변수의 스코프 이야기
 - 스코프 체이닝
 - 고차함수
- 다시 클로저로 돌아와서
- 언제써요?
- 주의점

클로저 발표배경

보통 개발자들의 인식

- 클로저? 일단 소스코드를 보자

```
#closure Python
def func():
    def funcInner():
        pass
    return funcInner
```

```
//closure javascript
function outerFunc(){
    console.log("out")
    return function (){
        console.log("in")
    }
}
```

- 함수 안에 함수가 있고 함수를 되돌려주는걸 클로저라고해
- 회사에서 잘 안써서 개념만 알고있어

들어가기전에 앞서서

- 변수의 스코프 이야기
- 스코프 체이닝
- 고차함수

※ 소스코드는 파이썬과 자바스크립트로 준비해 봤습니다

변수의 스코프 이야기

- 함수형 프로그래밍 에서의 변수 스코프 이야기
- 렉시컬 스코프
- 렉시컬 스코프?

검색!

names when functions are nested. The word "lexical" refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

한국어 ∨

높임말 

어휘적 범위 지정은 소스 코드 내에서 변수가 선언된 위치를 사용하여 해당 변수가 사용 가능한 위치를 결정한다는 사실을 의미합니다. 내포된 함수는 외부 범위에서 선언된 변수에 액세스할 수 있습니다.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

렉시컬 스코프 참고소스

```
//lexical scope (javascript)
var x = 10;

function foo() {
  var x = 1;
  bar();
}

function bar() {
  console.log(x);
}

foo(); // ?
bar(); // ?
```

```
#lexical scope(Python)
x = 10
def func1():
  x = 1
  func2()

def func2():
  print(x)

func1() #?
func2() #?
```

- 함수가 호출된 위치? 함수가 선언된 위치?
- 함수가 선언된 위치가 기준이 되어 변수를 참조하는 것

스코프 체이닝

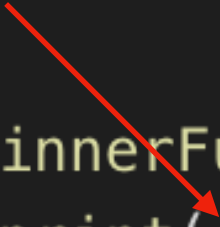
- 렉시컬 스코프를 이해하기 위한 단어
- 함수가 정의되어 있는 곳을 기준으로
- 변수를 찾는 과정
- 함수 내에서 이러한 작업이 일어났을때
- 내부에서 선언된 함수가 외부의 변수를
- 참조하는 행위를 스코프 체이닝

```
#scope chaining
def outerFunc():
    outerValue = 123

    def innerFunc():
        print(outerValue)

    innerFunc()

outerValue = 9998
outerFunc() # 123
```



고차함수

- 함수는 인자로 받을 수가 있다
- 함수가 리턴 될 수 있다

함수는 인자로 받을 수 있다

- jQuery
- 자바스크립트 콜백함수
- 파이썬 map filter reduce

```
//jQuery call back
$("#buttonA").click(function(){
    alert("you cleck button A")
})
```

```
//call back
function alertAbc(){
    alert("hello");
}
document.getElementById("abc").addEventListener("click", alertAbc())
```

```
print(list(map(lambda x : x * 2 , [2,4,6,8])))
```

함수가 리턴 될 수 있다

- 람다식을 이용하여 리턴을 할 수 있고
- 클로저(함수를 리턴)하는 식으로 구성
- 할 수 도 있습니다

```
#higher order function 고차함수 - 1
def outerFunc1(args):
    return lambda x: args + x

a=outerFunc1(1)
print(a(333)) # 334
```

다시 클로저로 돌아와서

클로저란?

- 함수가 리턴 한 값은 함수
- 함수내부의 스코프 와 관련이 있다
- 함수형 프로그래밍에 쓰인다고 한다

```
#closure Python
def func():
    def funcInner():
        pass
    return funcInner
```

```
//closure javascript
function outerFunc(){
    console.log("out")
    return function (){
        console.log("in")
    }
}
```

함수가 리턴 한 값은 ‘함수’

- 프로그래머 분들이 알고계시는 그 ‘형태’가 맞습니다

```
def makeHelloSentence(peopleName):  
    hi = "Hi " + peopleName  
    myNameIs = " my name is "  
  
    def makeASentence(myName):  
        return hi + myNameIs + myName + "."  
    return makeASentence  
  
sentence = makeHelloSentence("jonggu")  
print(sentence("seongbin"))
```

```
function makeHelloSentence(yourName){  
    let hi = "hi " + yourName;  
    let andSentence = " and "  
    let myNameIs = "my name is ";  
    return function inner(myName){  
        return hi + andSentence + myName  
    }  
}  
  
a = new makeHelloSentence("jonggu");  
console.log(a("seongbin"))
```

- 개념을 알려면 ‘형태’ 가지고는 안될것 같았습니다

처음 적용할 때...

```
def makeHelloSentence(peopleName):  
    hi = "Hi " + peopleName  
    myNameIs = " my name is "  
  
    def makeASentence(myName):  
        return hi + myNameIs + myName + "."  
    return makeASentence  
  
sentence = makeHelloSentence("jonggu")  
print(sentence("seongbin"))
```

함수를 반환



```
def iDontKnowClosure(param):  
  
    def innerFunc(param1):  
        return param + param1  
    return innerFunc(1)  
  
result = iDontKnowClosure(1)  
  
print(result)
```

결과값을 반환



자바의 결과값리턴과 헷갈리기도

- 자바는 return 한순간
- 함수의 변수들이 저장되어 있던 스택 메모리가 소멸되기 때문에
- 클로저(스코프의 보존)가 일어 날 수 없습니다

함수내부의 스코프 와 관련이 있다

- 스코프? 변수? 맞습니다!
- 현재의 변수 스코프는?
- 즉 sentence에서는

```
def makeHelloSentence(peopleName):  
    hi = "Hi " + peopleName  
    myNameIs = " my name is "  
  
    def makeASentence(myName):  
        return hi + myNameIs + myName + "."  
    return makeASentence  
  
sentence = makeHelloSentence("jonggu")  
print(sentence("seongbin"))
```

makeHelloSentence의 정보를 가지고있으면서

makeASentence 의 정보를 가지고있다는 점입니다

함수 내부의 스코프 이지만

- 함수가 끝나게 되면?
- 안에 있는 hi, myNameis
- 의 내용은 사라지지 않습니다
- 언제 어디에서 쓰일지 모르기 때문에 스코프가 유지 됩니다

```
def makeHelloSentence(peopleName):  
    hi = "Hi " + peopleName  
    myNameIs = " my name is "  
  
    def makeASentence(myName):  
        return hi + myNameIs + myName + "."  
    return makeASentence  
  
sentence = makeHelloSentence("jonggu")  
print(sentence("seongbin"))
```

언제쓰나요?

- 만들어진 함수를 수정할때
- 캡슐화
- 커링

만들어진 함수를 수정할때

- 수정해야 할 함수에 파라미터를 안바꾸고 결과값만 바꾸고 싶을때
- 내부에 함수를 만든 후 결과값으로 내보낼 수 있습니다

캡슐화

- Getter Setter는 자바만의 전유물이었는줄
- 알았는데 렉시컬 스코프가 지원되는
- 언어 에서도 쓸 수 있었습니다

```
//closure capsuleate

var capsuleate = function(){
  var num = 0;
  return {
    increse:function(){
      num++;
    },
    decrease:function(){
      num--;
    },
    getNum:function(){
      return num;
    }
  }
}

example1 = new capsuleate();
example1.increse();
example1.increse();
example1.decrease();
console.log(example1.getNum()) // 1

example2 = new capsuleate();
example2.decrease();
example2.decrease();
example2.decrease();
console.log(example2.getNum()); // -3
```

정리

- 함수가 함수를 반환하며
- 바깥쪽 함수의 환경을 저장했다가 안쪽 함수가 불러졌을때
- 저장한 환경을 이용하여 바깥쪽 함수의 루틴 실행
- 지금까지 배워본 것으로 비추어 볼 때 클로저는 마치 생성자의 역할을 하고 있었습니다

커링

인수의 부분 적용

지정된 수보다 적은 인수를 지정하는 경우 나머지 인수를 요구하는 새 함수를 만듭니다. 인수를 처리하는 이 메서드를 *커링* (currying)이라고 하며 F#과 같은 함수형 프로그래밍 언어의 특성입니다. 예를 들어, 두 가지 크기의 파이프(2.0의 반지름 및 3.0의 반지름)에 대해 작업하고 있다고 가정합니다. 다음과 같이 파이프의 볼륨을 결정하는 함수를 만들 수 있습니다.

함수 호출

함수 이름 뒤에 공백을 지정한 후 공백으로 구분된 인수를 지정하여 함수를 호출합니다. 예를 들어, **cylinderVolume** 함수를 호출하여 **vol** 값에 결과를 할당하려면 다음 코드를 작성합니다.

F#


복사

```
let vol = cylinderVolume 2.0 3.0
```

<https://docs.microsoft.com/ko-kr/dotnet/fsharp/language-reference/functions/>

커링

F#

 복사


```
let smallPipeRadius = 2.0
let bigPipeRadius = 3.0

// These define functions that take the length as a remaining
// argument:

let smallPipeVolume = cylinderVolume smallPipeRadius
let bigPipeVolume = cylinderVolume bigPipeRadius
```

다양한 길이의 두 가지 크기 파이프에 필요한 추가 인수를 제공합니다.

F#

 복사

```
let length1 = 30.0
let length2 = 40.0
let smallPipeVol1 = smallPipeVolume length1
let smallPipeVol2 = smallPipeVolume length2
let bigPipeVol1 = bigPipeVolume length1
let bigPipeVol2 = bigPipeVolume length2
```

커링

예를 들어, F# 의 경우, **forward pipe** 라고 부르는 `|>` 연산자를 지원한다. 다음의 F# 코드 예시를 살펴보자.

```
let res = [ 1 .. 10 ] |> List.filter (fun x -> x % 2 = 0) |> List.sum
printfn "Result : %i" res
```

첫 문장을 말로 풀어보면 다음과 같다.

1. 1부터 10까지의 수열 중
2. 2로 나눈 몫이 0인 것들만 추려내고
3. 모두 더한다

커링이 없다면

위 F# 예시에서 눈치챘을지 모르겠지만, **커링** 을 사용했다. 만약 F# 에서 커링을 지원해주지 않았다면 다음과 같이 코드가 바뀌어야 했을 것이다.

커링

- 클로저로 구현 할 수 있다

주의점

- 메모리 누수가 일어날 수 있습니다

QnA

감사합니다