CIS 61
Practice 03 - Higher Order Functions

Part 1 - What Would Python Display?

Type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

As a reminder, the following two lines of code will not display anything in the Python interpreter when executed:

```
>>> x = None
>>> x
```

Practice 1:

```
>>> lambda x: x  # A lambda expression with one parameter x


______
>>> a = lambda x: x  # Assigning the lambda function to the name a
>>> a(5)


______
>>> (lambda: 3)()  # Using a lambda expression as an operator in a call exp.


______
>>> b = lambda x: lambda: x  # Lambdas can return other lambdas!
>>> c = b(88)
>>> c


>>> z = 3
>>> e = lambda x: lambda y: lambda: x + y + z
>>> e(0)(1)()


______
>>> f = lambda z: x + z
>>> f(3)
```

Python 3.8.5 Shell

```
>>>
>>>
>>> lambda x: x
<function <lambda> at 0x10746cdc0>
>>>
>>> a = lambda x: x
>>> a(5)
5
>>>
>>> (lambda: 3)()
3
>>>
>>> b = lambda x: lambda: x
>>> c = b(88)
>>> c
<function <lambda>.<locals>.<lambda> at 0x107480040>
>>>
>>> z = 3
>>> e = lambda x: lambda y: lambda: x + y + z
>>> e(0)(1)()
4
>>>
>>> f = lambda z: x + z
>>> f(3)
Traceback (most recent call last):
  File "<pyshell#239>", line 1, in <module>
    f(3)
  File "<pyshell#238>", line 1, in <lambda>
    f = lambda z: x + z
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>>
```

Ln: 463 Col: 5

```
>>> higher_order_lambda = lambda f: lambda x: f(x)
>>> g = lambda x: x * x
>>> higher_order_lambda(2)(g)  # Which argument belongs to which function call?
```

(2) belongs to f(x), (g) belongs to lambda x: f(x)

```
>>> higher_order_lambda(g)(2)

______
>>> call_thrice = lambda f: lambda x: f(f(f(x)))
>>> call_thrice(lambda y: y + 1)(0)

______
>>> print_lambda = lambda z: print(z)  # When is the return expression of a lambda
expression executed?
>>> print_lambda

______
>>> one_thousand = print_lambda(1000)

______
>>> one_thousand

______
```

Python 3.8.5 Shell

```
>>> higher_order_lambda = lambda f: lambda x: f(x)
>>> g = lambda x: x * x
>>> higher_order_lambda(2)(g)
Traceback (most recent call last):
  File "<pyshell#265>", line 1, in <module>
    higher_order_lambda(2)(g)
  File "<pyshell#263>", line 1, in <lambda>
    higher_order_lambda = lambda f: lambda x: f(x)
TypeError: 'int' object is not callable
>>> higher_order_lambda(g)(2)
4
>>>
>>> call_thrice = lambda f: lambda x: f(f(f(x)))
>>> call_thrice(lambda y: y + 1)(0)
3
>>>
>>> print_lambda = lambda z: print(z)
>>> print_lambda
<function <lambda> at 0x107480790>
>>>
>>> one_thousand = print_lambda(1000)
1000
>>> one_thousand
>>>
>>>
```

Ln: 530 Col: 0

Practice 2: Type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

```
>>> def even(f):
...     def odd(x):
...         if x < 0:
...             return f(-x)
...         return f(x)
...     return odd
>>> steven = lambda x: x
>>> stewart = even(steven)
>>> stewart


______
>>> stewart(61)


______
>>> stewart(-4)


______
```

```
>>> def cake():
...   print('beets')
...   def pie():
...       print('sweets')
...       return 'cake'
...   return pie
>>> chocolate = cake()


______
>>> chocolate


______
>>> chocolate()


______
>>> more_chocolate, more_cake = chocolate(), cake


______
>>> more_chocolate


______
```

Python 3.8.5 Shell

```
>>> def even(f):
        def odd(x):
                if x < 0:
                        return f(-x)
                return f(x)
        return odd

>>> steven = lambda x: x
>>> stewart = even(steven)
>>> stewart
<function even.<locals>.odd at 0x1074808b0>
>>>
>>> stewart(61)
61
>>> stewart(-4)
4
>>>
```

Ln: 565 Col: 4

Python 3.8.5 Shell

```
>>> def cake():
        print('beets')
        def pie():
                print('sweets')
                return 'cake'
        return pie

>>> chocolate = cake()
beets
>>> chocolate
<function cake.<locals>.pie at 0x107480940>
>>> chocolate()
sweets
'cake'
>>> more_chocolate, more_cake = chocolate(), cake
sweets
>>> more_chocolate
'cake'
>>>
```

Ln: 585 Col: 4

```
>>> def snake(x, y):
...     if cake == more_cake:
...         return lambda: x + y
...     else:
...         return x + y
>>> snake(10, 20)


_______
>>> snake(10, 20)()


_______
>>> cake = 'cake'
>>> snake(10, 20)


_______
```

Python 3.8.5 Shell

```
>>>
>>> def snake(x, y):
        if cake == more_cake:
                return lambda: x + y
        else:
                return x + y


>>> snake(10, 20)
<function snake.<locals>.<lambda> at 0x107480a60>
>>> snake(10, 20)()
30
>>> cake = 'cake'
>>> snake(10, 20)
30
>>> |
```

Ln: 601 Col: 4

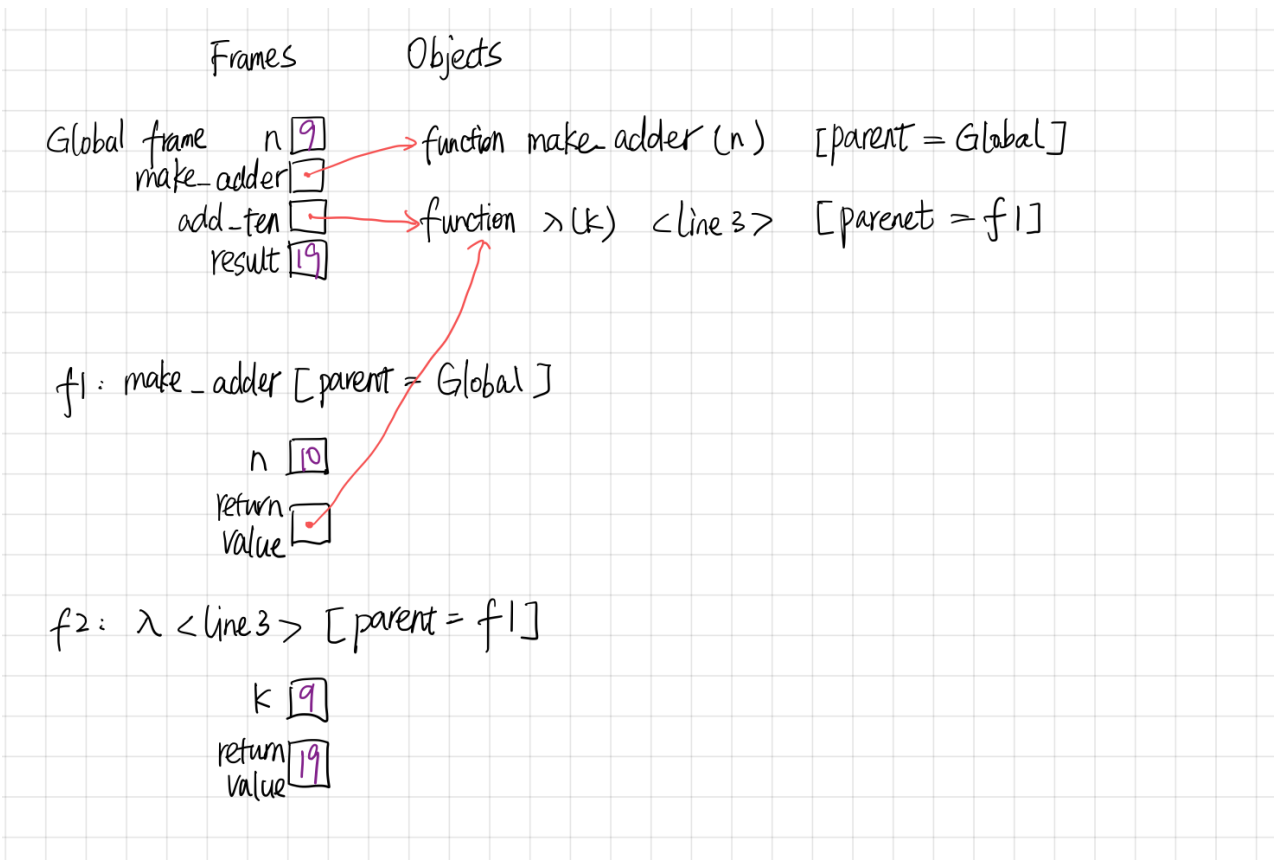
Part 2 - Environment Diagrams

I encourage you to do these problems on paper to develop familiarity with Environment Diagrams, which **will appear on the exam.**

You can check your work with the Online Python Tutor, but try drawing it yourself first

Practice 3 - Draw the environment diagram for the following code:

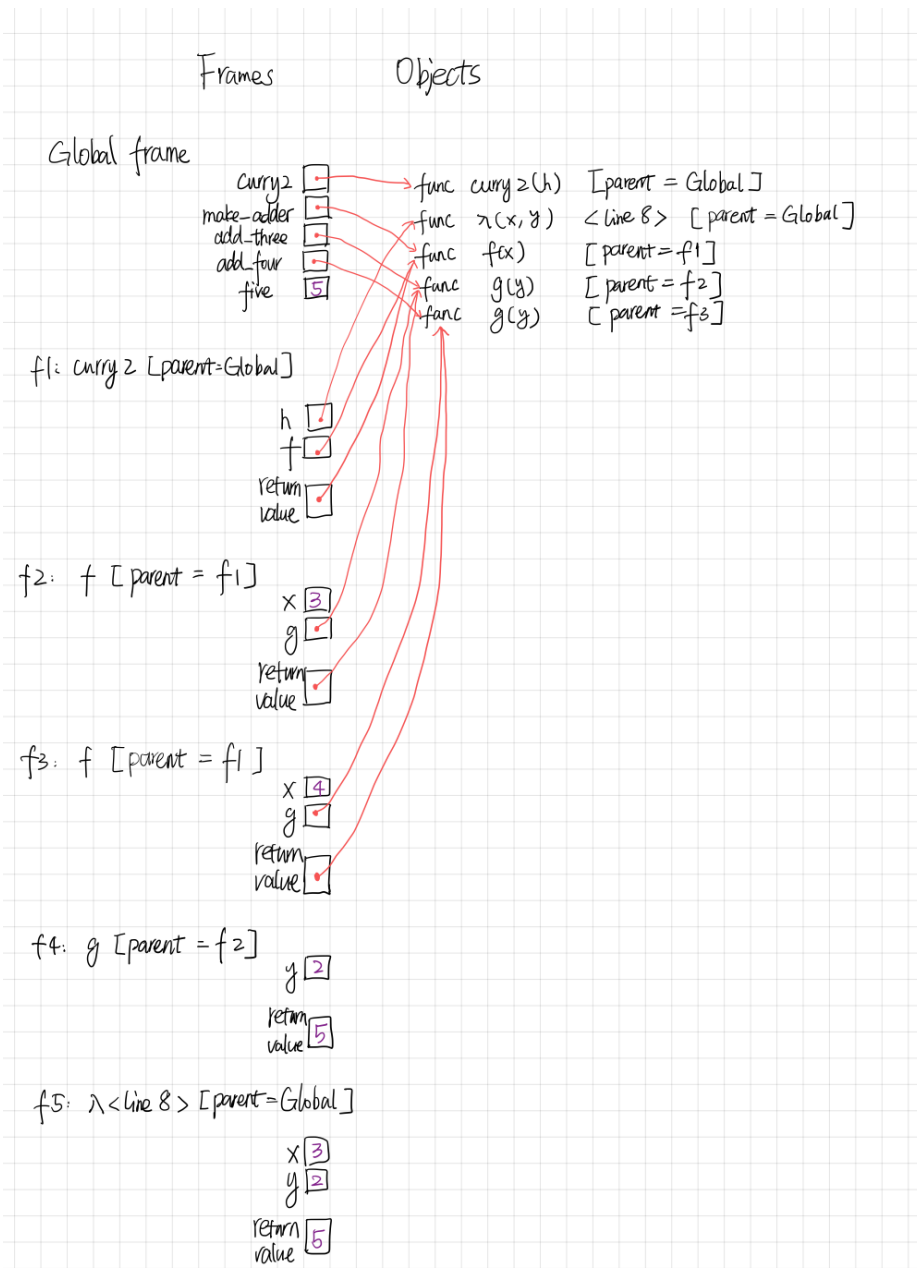
```
n = 9
def make_adder(n):
    return lambda k: k + n
add_ten = make_adder(n+1)
result = add_ten(n)
```



Practice 5 - Draw the environment diagram that results from executing the code below.

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(lambda x, y: x + y)
add_three = make_adder(3)
add_four = make_adder(4)
five = add_three(2)
```



Practice 6 - Draw the environment diagram that results from executing the code below.

```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)
g = (lambda y: y())(f)
```