Functional Closures and Decorators

Closures

- In order to understand closures, let's review the Python scoping rules: LEGB
  - L = local
  - E = enclosing
  - G = global
  - B = builtin (e.g., len() function)

In [1]:

```
a = 'global scope'

def outer_func():
    b = 'local to outer_func()'
    def inner_func():
        c = 'local to inner_func()'
        print(b, 'enclosing scope')
        print(a, 'global scope')
    inner_func()

outer_func()
```

```
local to outer_func() enclosing scope
global scope global scope
```

- When a function references a name that is not local, Python first attempts to resolve that name in the enclosing scope
- A *closure* is a nested function which remembers a value or values from the enclosing lexical scope even when the program flow is no longer in the enclosing scope

In [2]:

```
def make_adder(x):
    print('id(x): %x' % id(x))

    def adder(y):
        print('in adder')
        return x + y # Python uses LEGB to find 'x'

    print('id(adder): %x' % id(adder))
```

```python
    return adder

add39 = make_adder(39)
print('about to call add39')
add39(109)
```

```
id(x): 956180
id(adder): 7ff29c5de040
about to call add39
in adder
```

Out[2]:

```
148
```

In [3]:

```python
# Let's use repr so we can see the address of the function
# we could use print("%X") as well...
type(add39), repr(add39)
```

Out[3]:

```
(function, '<function make_adder.<locals>.adder at 0x7ff29c5de040>')
```

In [4]:

```python
# all functions have a closure attribute
add39.__closure__
```

Out[4]:

```
(<cell at 0x7ff29c5e1190: int object at 0x956180>,)
```

In [5]:

```python
# notice that the cell object has a reference to an int object
add39.__closure__[0].cell_contents
```

Out[5]:

```
39
```

In [6]:

```python
print(make_adder.__closure__)
```

```
None
```

- One case where closures are frequently used is in building function wrappers
- Suppose we want to log each invocation of a function:

In [7]:

```python
def logging(f):
    def wrapper(*args, **kwargs):
        print('Calling %r(%r, %r)' % (f, args, kwargs))
```

```python
        return f(*args, **kwargs)
    return wrapper
```

In [8]:

```python
logging_add39 = logging(add39)
print(add39(5)) # remember that add39 just adds 39 to our argument
```

```
in adder
44
```

In [9]:

```python
print(logging_add39(5))
```

```
Calling <function make_adder.<locals>.adder at 0x7ff29c5de040>((5,), {})
in adder
44
```

In [10]:

```python
logging_add39.__closure__[0].cell_contents
```

Out[10]:

```
<function __main__.make_adder.<locals>.adder(y)>
```

Decorators

- Wrapper functions are so common, that Python has its own term for it–a *decorator*.
- Why might you want to use a decorator?
    - sometimes you want to modify a function's behavior without explicitly modifying the function, e.g., pre/post actions, debugging, etc.
    - suppose we have a set of tasks that need to be performed by many different functions, e.g.,
        - access control
        - cleanup
        - error handling
        - logging
    - ...in other words, there is some boilerplate code that needs to be executed before or after every invocation of the function

Decorators build on topics we already know...

- nested functions
- variable positional args (`*args`)

- variable keyword args (**kwargs)
- functions are objects (actually everything in Python is an object)

In [11]:

```
def document_it(func):
    # below is a nested, or inner function
    def new_function(*args, **kwargs):
        print(f'Running function: {func.__name__}')
        print(f'Positional arguments: {args}')
        print(f'Keyword arguments: {kwargs}')
        # here we invoke the function passed in as an argument
        result = func(*args, **kwargs)
        print(f'Result: {result}')
        return result

    # document_it() is returning a reference to the inner function
    return new_function
```

In [12]:

```
def add_things(a, b):
    return a + b

print('Running plain old add_things()')
print(add_things(13, 5))
```

```
Running plain old add_things()
18
```

In [13]:

```
# manual decorator assignment
cooler_add_things = document_it(add_things)

print('Running cooler_add_things()')
cooler_add_things(13, 5)
```

```
Running cooler_add_things()
Running function: add_things
Positional arguments: (13, 5)
Keyword arguments: {}
Result: 18
```

Out[13]:

```
18
```

In [14]:

```
# decorator shorthand for what we did above
```

```
#@document_it
def add_things(a, b):
    return a + b

#add_things = document_it(add_things)

print(add_things(13, -5))
```

```
8
```

In [15]:

```
print(id(add_things))
add_things = document_it(add_things)
print(add_things(13, -5))
print(id(add_things))
```

```
140679982505552
Running function: add_things
Positional arguments: (13, -5)
Keyword arguments: {}
Result: 8
8
140679982211424
```

Lab: Decorators

1. Create a function called **`printer`** that takes a string and prints it
   - Then create a wrapper that will print the number of times each letter appears in the string passed in to **`printer`**, followed by the string.
   - Use the wrapper as a decorator on your **`printer`** function.
2. Create some function which takes an integer as its parameter
   - Create a wrapper that ensures the parameter is positive
   - use that wrapper to decorate your original function
3. Make a timer decorator that computes the elapsed time of the function wrapped by it