Functional Programming

Pure Function

Pure Function - accepts some arguments & returns some results (output) based on the arguments.

Given the same argument values, a pure function always returns the same result (output).

Invoking a pure function does **not** have side effects.

Example: `math.sqrt`

```
r = math.sqrt(x)
```

1. result (output) depends only only x
2. if x is same, the result is always the same
3. does not change anything else (no "side effects")

Non-pure Function

Non-pure Function - any function that is not pure.

- same arguments (input) can produce different outputs
- may have an output without any inputs
- may have side effects, such as changing state of some component in the program

Give Examples in the Python Library

1. A function that returns different outputs for the same inputs, or no inputs.

2. A function that has a side-effect on the environment.

3. A function that changes the state of a part of the program.

Environment & Bindings

The **environment** contains all the names that are known at a given point in the program execution.

The environment consists of frames.

A frame contains a symbol table of known names and their values.

These are called bindings.

```
from math import  sqrt
x = 25
y = sqrt(x)
```

<u>Frame</u>

sqrt	<function>
x	25
y	5

Environment Contains Multiple Frames

When f() is active, it knows all the names in its own frame, plus the names in the global frame.

```
from math import sqrt
x = 25
y = sqrt(x)
f(x)


def f(arg):
    sum = x + y
    print("x=", x)
    print("y=", y)
```

Global Frame

Name	Scope	Value
sqrt	file	λ(x): (math.sqrt)
x	file	25
y	file	5
f	file	λ(x): ...

f Frame

Name	Scope	Value
arg	f	25
sum	f	30

Local Variables have Local Scope

A variable defined inside a function is visible only inside that function

```
x = 25
y = 5
g()


def g():
    x = 10
    f(y-1)


def f(y):
    sum = x + y
    print("x=", x)
    print("y=", y)
```

Global Frame

Name	Scope	Value
x	file	25
y	file	5
f	file	λ(x): ...
g	file	λ(): ...

g Frame

Name	Scope	Value
x	g	10

f Frame

Name	Scope	Value
y	f	what?
sum	f	what?

Functional Programming by Example

Example Problem

Define a function to compute and return each of these:

- sum of first n positive integers
- sum of squares of first n positive integers
- sum of cubes of first n positive integers
- sum of inverse (1/k) of first n positive integers

Boring, Redundant Solution

```
def sum_ints(n):
    total = sum(range(1,n+1))
    return total

def sum_squares(n):
    return sum(k*k for k in range(1,n+1))

def sum_cubes(n):
    return sum(k*k*k for k in range(1,n+1))

def sum_inverse(n):
    return sum(1/k for k in range(1,n+1))
```

Function as Parameter

In Python, you can pass a function as a parameter:

```
def eval_and_print(fun, x):
    fname = fun.__name__
    print(f"{fname}({x}) =", fun(x))

>>> import math
>>> eval_and_print(math.sqrt, 5)
sqrt(5) = 2.2360679775
>>> eval_and_print(math.log10, 1000)
log10(1000) = 3.0
>>> eval_and_print(math.exp, 1)
exp(1) = 2.7182818284
```

Less Redundant Solution

```
def sum_of(fun, n):
    """sum fun(k) for first n positive ints"""
    return sum(fun(k) for k in range(1,n+1))

def identity(x):
    return x
def square(x):
    return x*x
def inverse(x):
    return 1/x

sum_of(identity, 100)
sum_of(square, 100)
sum_of(inverse, 100)
```

Function as Return Value

A function can return a function.

```
def powerfun(n: int):
    """return a new function that computes
    n-th power of a single parameter."""
    def fun(x):
        return math.pow(x, n)
    return fun

>>> cube = powerfun(3)
>>> cube(10)
1000.0
>>> inverse = powerfun(-1)
>>> inverse(5)
0.2
```

Try it Yourself!

Define `powerfun(n)`.

Use `powerfun` to define and then verify each of these:

1. Create your own quad-power function = x^4
2. Create your own sqrt function
3. Create your own cube root function = $x^{1/3}$

```
>>> quad = powerfun(4)
>>> quad(10)
10000.0
# You can create a function and use it
# in the same statement
>>> powerfun(-1)(5)
0.2
```

Use powerfun in the sum_of problem

```
def sum_of(fun, n):
    """sum fun(k) for first n positive ints"""
    return sum(fun(k) for k in range(1,n+1))


sum_of(powerfun(1), 100)    # 1 + 2 + ... + 100
sum_of(powerfun(2), 100)    # sum of squares
sum_of(powerfun(3), 100)    # sum of cubes
inverse = powerfun(-1)
sum_of(inverse, 100)        # 1 + 1/2 +...+ 1/100
```

Lambda: anonymous functions

Lambda defines a new function as a single statement.

Syntax:

```
fun = lambda arguments: expression
```

```
>>> square = lambda x:  x * x
>>> square(5)
25

>>> signum = lambda x: -1 if x < 0 else 1
>>> signum(12)
1
>>> signum(-5)
-1
```

Use lambda instead of powerfun

```
def sum_of(fun, n):
    """sum fun(k) for first n positive ints"""
    return sum(fun(k) for k in range(1,n+1))

# Define a lambda for identity function f(x)= x
identity = lambda ___________________________
sum_of(identity, 100)     # 1 + 2 + ... + 100

# Define a lambda for inverse function (1/x)
# and use it, all in one statement
sum_of( __________ , 100) # 1 + 1/2 + ...+ 1/100
```

Functions as First Class Entities

1. A function can be used as a parameter

```
def sum_of(fun, n)
```

2. A function can create & return another function

```
def powerfun(n):
    ...
    return fun
```

3. A function can be assigned to a variable for later use.

```
square = powerfun(2)
inverse = lambda x: 1 / x
```

4. `lambda` defines a new (nameless) function

Higher-Order Functions

A *higher order function* is a function that accepts a function as input (parameter) and/or returns a function as a result.

Section 1.6 of *Composing Programs* covers higher order functions.

Function Remembers its Surrounding Environment

The cube function remembers that n = 3.

```
def powerfun(n: int):
    def fun(x):
        return math.pow(x, n)
    return fun

cube = powerfun(3)        # n = 3
root = powerfun(0.5)      # n = 0.5

cube(5)
```

```
fun(5):
    return math.pow(5,n)
```

Composing functions

Function composition means to define a new function as a combination of other functions (f and g):

$$h = f * g$$

means

$$h(x) = f(\ g(x)\ )$$

Can we **compose** two functions to define 1/(x*x)?

```
square = powerfun(2)       # square(5) is 25
inverse = lambda x: 1/x   # inverse(x) =1/x
```

Unfortunately, this doesn't work:

```
inverse_square = inverse(square)
inverse_square(5)    # should be 1/25
```

Exercise: define compose(f, g)

Define a function named **compose** that has 2 functions as parameters (f and g) and returns a new function that is the composition of f and g.

$$h = \text{compose}(f, g)$$

means $h(x) = f(\ g(x)\ )$

```
def compose(f, g):
    # TODO

### test
inverse = lambda x: 1/x
square =  powerfun(2)    # or lambda x: x*x
isquare = compose(inverse, square)
isquare(10)              # should be 0.01
```

Decorators

A *decorator* in Python is something that augments or adds functionality to another function. It is written as:

```
@decorator
def some_function(args):
    ...
```

Next time: how to define your own decorators.

Tools for Functional Programming

The Python **`functools`** library contains function decorators and utilities for higher-order functions.

Example:

1. Run the naive fibonacci function from lab 2.

```
def fibonacci(n: int):
    if n <= 0: return 0
    if n == 1: return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Run it with n > 32. It is terribly slow.

Now try this...

functools lru_cache

```
from functools import lru_cache

@lru_cache
def fibonacci(n: int):
    if n <= 0: return 0
    if n == 1: return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Run it again.
Notice any difference?

References

Composing Programs
Section 1.3 - environments
Section 1.6 - higher order functions
https://composingprograms.com/

Functional Programming in Python on realpython.com
Covers the important map-reduce concept
https://realpython.com/python-functional-programming/