Functional Programming

- Functional programming in Python
- Higher order functions
- Additional techniques

1. Functional Programming in Python

- Overview of functional programming (FP)
- Function evaluation
- Pure functions
- Anonymous functions a.k.a. lambdas
- Lambda example
- Lambdas and parameters

Overview of functional programming (FP)

FP is a style of programming characterised by…

- Treating computation as the evaluation of functions
- Use of higher-order functions and/or recursion
- Immutable (read-only) state
- Lazy evaluation

Why use FP?

- Very amenable to multi-threading
- Share complex algorithms across multiple threads, to maximise concurrency and increase performance

Any disadvantages?

- Quite a steep learning curve
- Not suitable for every problem

Function evaluation

Functions depend only on their inputs, and not on other program state

For example, consider the following function

```python
def cube(x):
  return x * x * x
```

It always gives the same answer for the same input (so it has predictable behaviour - you can reason about its operation)

It has no local state, side-effects, or changes to any other program state (so it can be safely executed by multiple threads)

Pure functions

A "pure function" is one that has no side effects. This has several useful consequences:

- If the result of a pure expression is not used, it can be removed without affecting anything else
- If a pure function is called with the same arguments, you will get the same result (so evaluations can be cached)
- If there is no data dependency between two pure functions, they can be evaluated in any order, or performed in parallel

Anonymous functions a.k.a. lambdas

A lambda expression is a 1-line inline expression

- Like an anonymous function

To define a lambda expression:

- Use the lambda keyword
- Followed by the argument list
- Followed by a colon
- Followed by a 1-line inline expression

```
my_lambda = lambda arg1, arg2, … argn  :  inline_expression
```

To invoke a lambda expression:

- Same syntax as a regular function call

```
my_lambda(argvalue1, argvalue2, …, argvaluen)
```

Lambda example

A lambda that takes a single parameter and returns the square of that value

```
mylambda = lambda x: x * x

result = mylambda(10)
print(result)
```

Lambdas and parameters

Lambdas can take multiple parameters

- List all the parameters after the lambda keyword

```
mylambda = lambda x, y: print(f"You passed {x}, {y}")
mylambda(10, 20)
```

Lambdas can take no parameters

- Just follow the lambda keyword with a : immediately

```
mylambda = lambda: print("Hello!")

mylambda()
```

2. Higher Order Functions

- Overview of higher-order functions
- Passing a lambda to a function
- Returning a lambda from a function
- Closures

Overview of higher-order functions

Higher-order functions can use other functions as arguments and return values

- You can pass a function as a parameter into another function
- You can return a function from a function

We'll explore both these techniques in the following slides

- We'll use lambdas to represent the function parameters/returns

Passing a lambda to a function

You can pass a lambda as a parameter into a function

- Allows you to write very generic functions

Example

- The apply() function applies the lambda that you pass in

```
def apply(arg1, arg2, op) :
    return op(arg1, arg2)

result1 = apply(10, 20, lambda x, y: x + y)
print(result1)

result2 = apply(10, 20, lambda x, y: x / y)
print(result2)
```

Returning a lambda from a function

You can return a lambda from a function

Consider this simple concat() function

- Concatenates its two parameters in the order specified

```
def concat(str1, str2):
    return str1 + str2
```

Now consider the flip() function

- Takes a binary operation
- Returns a lambda that performs the operation with args flipped

```
def flip(binaryOp) :
    return lambda x, y: binaryOp(y, x)

# Usage.
flipConcat = flip(concat)
result2 = flipConcat("Hello", "World")
print(result2)
```

Closures (1/2)

A closure is a function whose behaviour depends on variables declared outside the scope in which it is then used

- This is often used when returning functions/lambdas
- The returned function/lambda remembers the original state in the enclosing function

```
def banner(start, end) :
    return lambda msg: print(f"{start} {message} {end}")

bannerMsg = banner("[---", "---]")

bannerMsg("Hello")
bannerMsg("World")
```

Closures (2/2)

Here, fib returns a function that calculates Fibonacci numbers, returns the next one each time called

```python
def fib():
    tup = (1,-1)
    def retfunc():
        nonlocal tup
        tup = (tup[0] + tup[1], tup[0])
        return tup[0]

    return retfunc
```

Note 1: nonlocal keyword lets you access a variable in external scope

Note 2: tup is a tuple, and you access its members using [0] and [1]

3. Additional Techniques

- Recursion
- Tail recursion
- Reduction
- Partial functions

Recursion

Recursion is commonly used instead of looping

- It avoids the mutable state associated with loop counters

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

```
result = factorial(4)
print("4 factorial is %d\n" % result)
```

Tail Recursion

Tail recursion is where the very last thing you do in a function is call yourself

- The function calls can theoretically be executed in a simple loop

Here's a tail-recursive implementation of factorial

```
def tailRecursiveFactorial(accumulator, n):
    if n == 0:
        return accumulator
    else:
        return tailRecursiveFactorial(n * accumulator, n - 1)

result = tailRecursiveFactorial(1, 4)
print("4 factorial is %d\n" % result)
```

Reduction

The functools module has several useful utility functions for functional programming

- E.g. reduce(), which reduces the elements in a collection to a single result

```
from functools import reduce

mylambda = lambda x,y: x+y

result = reduce(mylambda, [3,12,19,1,2,7])
print(result)
```

Partial Functions

The functools module also allows you to create partial functions, i.e. functions with one or more args already filled in

- Via partial()

```
from functools import partial

multiply = lambda x,y: x * y

times2 = partial(multiply, 2)
times5 = partial(multiply, 5)
times8 = partial(multiply, 8)

print("10 times 2 is %d" % times2(10))
print("10 times 5 is %d" % times5(10))
```

Note: If you're interested to learn how this works, see our own version of partial() here:

- PartialFunctionsHowTheyWork.py

Any Questions?