The UK's European university

University of Kent

COMP8270 / PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Fernando Otero
febo@kent.ac.uk
cs.kent.ac.uk/people/staff/febo

overview:

1. Default Arguments
2. Lambda Closures
3. Functions as Objects

Default Arguments

Arguments to Functions (so far)

- Positional matching
  - `def f(a, b, c)`
  - invoke with `f(x, y, z)`
- Optional number of arguments
  - `f(*argument_list)`
- KWARGS: Keyword arguments
  - `f(**dictionary)`
- We will now discuss:
  - name matching
  - default arguments

Arguments: Name Matching

```
def f(x, y, z):
        return x + y + z

f (1, 2, 3)                          # Legal
f (x = 1, y = 2, z = 3)              # Legal
f (1, 2, z = 3)                      # Legal
f (z = 3, 1, 2)                      # Error
f (1, z = 3, y = 2)                  # Legal
```

- Python can match argument names.
- All name matching must follow the position arguments (to avoid ambiguity).
- Ask yourself how this is materially different from KWARGS.

Default Arguments

```
def f(x = 1, y = 2, z = 3):
        return x + y + z

f ()                                    # Legal
f (x = 5)                               # Legal
f (y = 5)                               # Legal
f (z = 5)                               # Legal
f (z = 1, y = 1)                        # Legal
f (1, 2)                                # Legal
f (1, z = 4)                            # Legal
f (y = 1, 1, 2)                         # Error
```

- Python supports default arguments.
- Include values in the function header.
- All positional arguments must precede named arguments.
  - No different from name matching.

Python Lambdas

λ Closures: Overview

- There are similar features in many languages.
- Originally derived from the λ calculus.
  - For the interested, a theory by Alonzo Church. (no state)
  - Turing's machine came later. (stateful)
- Popularized in LISP.
- In Python they are also known as anonymous "functions".

Lambda Expressions in Python

- Think of them as a *trivial* anonymous function.
  - But they are NOT functions.
- Light-weight: they are useful when one requires simple logic, but a function is not clearly indicated.
- They are often used as arguments to functions.
  - That is probably their only appropriate use.
- The general form is:

```
lambda argument_1, …, argument_N: statement
```

Python `filter` (1)

```
Z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filter (lambda x: x % 2 == 0, Z)
➞[2, 4, 6, 8, 10]
```

- This Python library function applies the operator to our list.
- It looks like: `filter(λ, iterable)`
  - This is a very common set of arguments.
- The lambda is applied to every element of the list.
- The lambda is (sort of) an anonymous function.
  - It has no name.

Python `filter` (1) con't

```
def Even(x):
        return x % 2 == 0


Z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filter(Even, Z)
➞[2, 4, 6, 8, 10]
```

- We could use a function, but it is rather heavy weight.
- The `filter()` returns a filter object. To view the contents convert it to a list:

```
list(filter(Even, Z))
```

Lambda Expressions in Python

- They do not use `return`.
- They do not support statements:
  - `for, try, if` etc. are not available
  - One cannot create new identifiers.
- They can be named with assignment:
  - `f = lambda x : x ** 2`
  - Invoke it with: `f(x)`
  - All the drawbacks of a function without any of the benefits.
  - Makes debugging more difficult (obfuscates the backtrace).
- Be careful when writing them. They can be very dense and difficult to read.

Python `filter` (2)

```
Z = [[1, 2], [3, 4], [5, 6]]
filter(lambda x: x[0] + x[1] < 10, Z)
➞ [[1, 2], [3, 4]]
```

- We can filter any iterable type (e.g. a list, dictionary), and they in turn may contain elements of any type.
- The implementor is responsible for understanding the type and doing something semantically sensible.
- In this example we are filtering based on an L1 distance.
- We can do anything we like!

The Lambda Idiom

- Polymorphic logic often needs a type specific operator:
  - A QuickSort implementation needs a total ordering (≤)
  - A clustering algorithm needs a distance metric ($|\cdot|$)
- Python has many functions that apply a lambda to a list: `map, filter, reduce, sum`,…
- Idiomatic Python employs such functions to increase speed.
- There are also faster mechanisms, but we will discuss them in later lectures.

Python `map`

```
Z = [[1, 2], [3, 4], [5, 6]]
list(map(lambda x: x[0]**3 + x[1]**3, Z))
➞ [9, 91, 341]
```

- We do can more than filter. There are many options.
- `map(λ, iterable)`
- Every element of the list is mapped to a new value.
  - Here we map each element, (x, y) ➞ x^3+y^3 , to a scalar.
- We can do whatever we like!

Another use of lambda: `reduce`

```
from functools import reduce
z = [1, 2, 3, 4, 5, 6, 7, 8, 9]
sum (z)
➞ 45
reduce(lambda arg, cumulative: arg + cumulative, z)
➞ 45
reduce(lambda arg, cumulative: arg + cumulative, z, 10)
➞ 55
```

- The `reduce` function is a cumulative function.
- It goes through the list and passes the previous result to the current computation.

Functions as Objects

Functions Are First Class Objects

```
def Identifier(Arguments):
            .
            .
            .
      return Something
```

- Unlike compiled languages, the identifier does not exist until the `def` is *executed*.
- Python creates a function object; it is a valid RVALUE.

```
AnotherName = Identifier
AnotherName(Arguments)
```

def at Run-Time

```
def Silly(x):
      if x < 0:
            def f():
                  print (“Less than zero”)
                  return
      else:
            def f():
                  print (“Not negative”)
                  return
      f()
      return
```

Or We Can Return the Function

```
def Silly(x):
      if x < 0:
            def f():
                  print (“Less than zero”)
                  return
      else:
            def f():
                  print (“Not negative”)
                  return
      return f

g = Silly(-1)
g()
Less than zero
```

Or We Can Pass It as an Argument

```
def Go(f):
      f()


g = Silly(-1)
Go(g)
> Less than zero
> MyCrazyList = [“Hello”, g, Go, 21]
> MyCrazyList[1]()
> Less than zero
```

Next lecture:

- **Object-Oriented Programming**