Functions and Functional Programming

DS 8015

Outline

Basic functions

RECALL

```
# The def keyword is used to define a new function

def fn_name(param1, param2):
    value = do_something()
    return value
```

RETURN

- All functions return some value

  ⇒ Even if that value is None

- No return statement or just return implicitly returns None

  ⇒ The interpreter suppresses printing None

- Returning multiple values

  ⇒ You can use a tuple! In some cases, use a namedtuple

  ```
  return value1, value2, value3
  ```

  ⇒ Be careful! Callers may not expect a tuple as a return value

Function Execution and Scopes

- Function execution introduces a new local symbol table (scope)

  $\Rightarrow$ Think of baggage tags and suitcases: a new baggage area

- Variable assignments (L-values) x = 5

  $\Rightarrow$ Add entry to local symbol table (or overwrite an existing entry)

- Variable references (R-values) print(y)
  - First, look in local symbol table
  - Next, check symbol tables of enclosing functions (unusual)
  - Then, search global (top-level) symbol table
  - Finally, check builtin symbols (print, input, etc)

Local Function Scope - 1

```
x = 2
def foo(y):
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)
# prints {'y': 3, 'z': 5}
# print 2
# prints 2, 3, 5
```

Local Function Scope - 2

```
x = 2
def foo(y):
    x = 41
    # We've added an 'x': 41 entry
    #  to the local symbol table
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)
# prints {'x': 41, 'y': 3, 'z': 5}
# print 2
# prints 41, 3, 5
```

IF / FOR SCOPE

- Notably, only function definitions (and classes) define new scopes
- if statements, for loops, while loops, with statements, etc do not introduce a new scope

```
if success:
    desc = 'Winner!'
else:
    desc = 'Loser :('
print(desc)
```

Default Parameters

Default / Named Parameters

- Specify a default value for one or more parameters
  $\Rightarrow$ Called with fewer arguments than it is defined to allow
- Usually used to provide "settings" for the function
- Why?
  - Presents a simplified interface for a function
  - Provides reasonable defaults for parameters
  - Declares intent to caller that parameters are "extra"

```
def ask_yn(prompt, retries=4, complaint='...'):

# Required param 'prompt'
# Optional param 'retries' default to 4
# Optional param 'complaint' defaults to some message
```

KEYWORD ARGUMENTS

```
def ask_yn(prompt, retries=4, complaint='Enter Y/N!'):
    for i in range(retries):
        ok = input(prompt)
        if ok in ('Y', 'y'):
            return True
        if ok in ('N', 'n'):
            return False
        print(complaint)
    return False

# Call with only the mandatory argument
ask_yn('Really quit?')
# Call with one keyword argument
ask_yn('OK to overwrite the file?', retries=2)
# Call with one keyword argument - in any order!
ask_yn('Update status?', complaint='Just Y/N')
# Call with all of the keyword arguments
ask_yn('Send text?', retries=2, complaint='Y/N please!')
```

Dead Parrot

```
def parrot(voltage, state='a stiff', action='voom',
    type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")

# 1 positional argument
parrot(1000)
# 1 keyword argument
parrot(voltage=1000)
# 2 keyword arguments
parrot(voltage=1000000, action='VOOOOOM')
# 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)
# 3 positional arguments
parrot('a million', 'bereft of life', 'jump')
# 1 positional, 1 keyword
parrot('a thousand', state='pushing up the daisies')
```

Invalid Calls

```
def parrot(voltage, state='...', action='...', type='...')

# required argument missing
parrot()
# non-keyword argument after a keyword argument
parrot(voltage=5.0, 'dead')
# duplicate value for the same argument
parrot(110, voltage=220)
# unknown keyword argument
parrot(actor='John Cleese')
```

Rules about Function Calls

- Keyword arguments must follow positional arguments
- All keyword arguments must identify some parameter

  $\Rightarrow$ Even positional ones
- No parameter may receive a value more than once

```
def fn(a): print(a)
fn(0, a=0)
# Not allowed
# multiples values of 'a'
```

Variadic Positional Arguments

VARIADIC POSITIONAL ARGUMENTS - 1

- A parameter of form *args captures excess positional args

  ⇒ These excess arguments are bundled into an args tuple

- Why?
  - Call functions with any number of positional arguments
  - Capture all arguments to forward to another handler

    ⇒ Used in subclasses, proxies, and decorators

```
print(*objects, sep=' ', end='\n', file=..., flush=False)
```

VARIADIC POSITIONAL ARGUMENTS - 2

```
# Suppose we want a product function that works as so:
product(3, 5) # => 15
product(3, 4, 2) # => 24
product(3, 5, scale=10) # => 150

# product accepts any number of arguments
def product(*nums, scale=1):
    p = scale
    for n in nums:
        p *= n
    return p

#Named parameters after *args are
# 'keyword-only' arguments
```

UNPACKING VARIADIC POSITIONAL ARGUMENTS

```
# Suppose we want to find 2 * 3 * 5 * 7 * ... up to 100
def is_prime(n): pass # Some implementation

# Extract all the primes
primes = [number for number in range(2, 100)
    if is_prime(number)]

# primes == [2, 3, 5, ...]
print(product(*primes)) # equiv. to product(2, 3, 5, ...)

#The syntax *seq unpacks a sequence
# into its constituent components
```

Variadic Keyword Arguments

VARIADIC KEYWORD ARGUMENTS - 1

- A parameter of the form **kwargs captures all excess keyword arguments

  $\Rightarrow$ These excess arguments are bundled into a kwargs dict

- Why?
  - Allow arbitrary named parameters, usually for configuration
  - Similar: capture all arguments to forward to another handler

    $\Rightarrow$ Used in subclasses, proxies, and decorators

VARIADIC KEYWORD ARGUMENTS - 2

```
authorize(
    "If music be the food of love, play on.",
    playwright="Shakespeare",
    act=1,
    scene=1,
    speaker="Duke Orsino"
)

# > If music be the food of love, play on.
# ----------------------------------------
# act: 1
# scene: 1
# speaker: Duke Orsino
# playwright: Shakespeare
```

VARIADIC KEYWORD ARGUMENTS - 3

```
def authorize(quote, **speaker_info):
    print(">", quote)
    print("-" * (len(quote) + 2))
    for k, v in speaker_info.items():
        print(k, v, sep=': ')


speaker_info = {
    'act': 1,
    'scene': 1,
    'speaker': "Duke Orsino",
    'playwright': "Shakespeare"
}
```

UNPACKING VARIADIC KEYWORD ARGUMENTS

```
info = {
    'sonnet': 18,
    'line': 1,
    'author': "Shakespeare"
}

authorize("Shall I compare thee to a summer's day", **info)

# > Shall I compare thee to a summer's day
# ----------------------------------------
# line: 1
# sonnet: 18
# author: Shakespeare
```

Complicated Example

```
"{0}{b}{1}{a}{0}{2}".format(
    5, 8, 9, a='z', b='x'
)
# => 5x8z59

# args = (5, 8, 9)
# kwargs = {'a':'z', 'b':'x'}
```

CUTE TRICK: UNPACKING VARIADIC KEYWORD ARGUMENTS

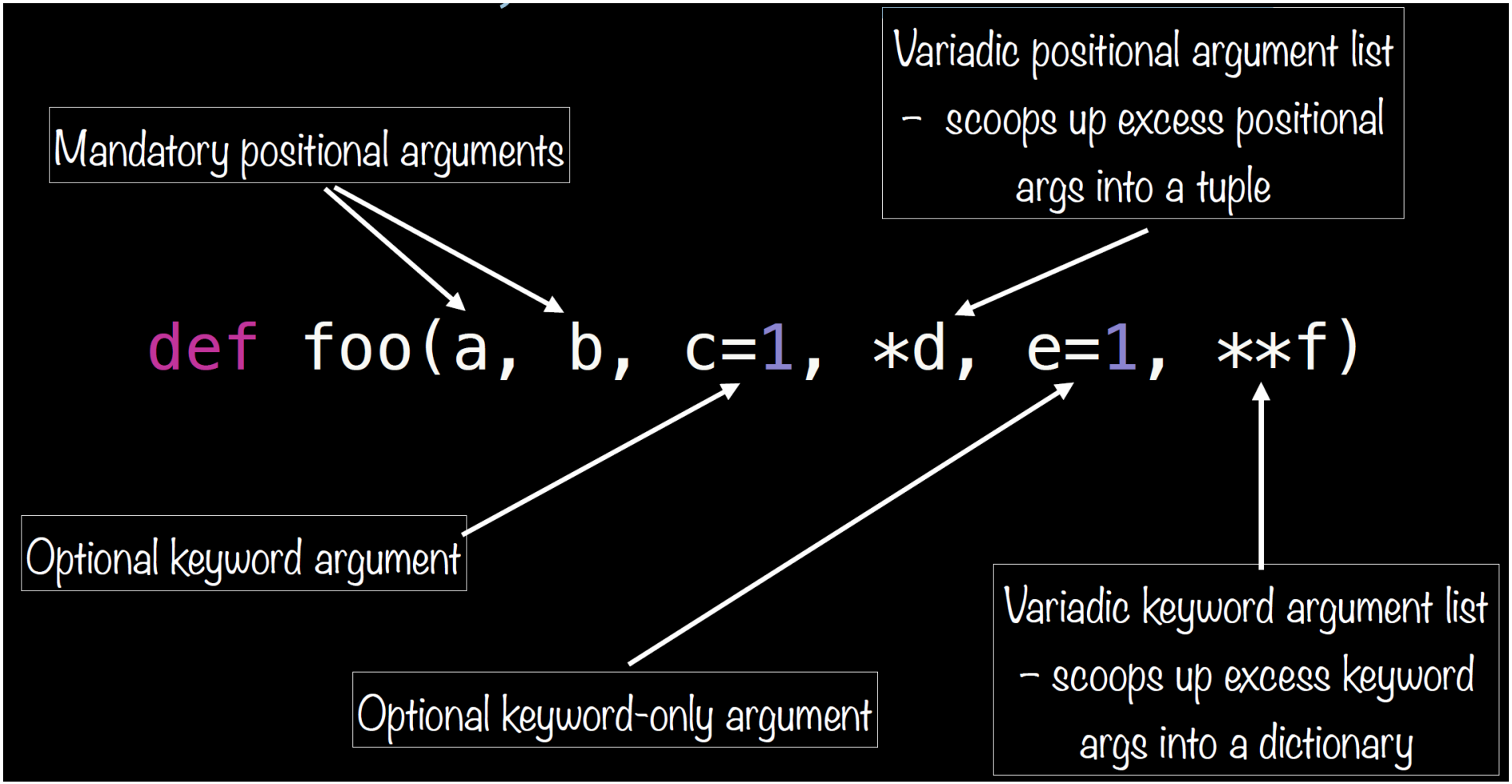
```
x = 3
foo = 'fighter'
y = 4
learn = 2, 'fly'
z = 5

local_symbol_table = {
'x': 3,
'foo': 'fighter',
'y': 4,
'learn': (2, 'fly'),
'z': 5, ...
}

print("{z}^2 = {x}^2 + {y}^2".format(x=x, y=y, z=z))
print("{z}^2 = {x}^2 + {y}^2".format(**locals()))
# Equivalent to .format(x=3, foo='fighter', y=4, ...)
```

PUTTING IT ALL TOGETHER

A Valid Python Function Definition:



SUMMARY

- All functions return some value (possibly None)
- Functions define scopes via symbol tables
- Parameters are passed by object reference
- Functions can have optional keyword arguments
- Functions can take a variable number of args and kwargs
- Use docstrings and good style
- Functions are objects too (?!)

WHY FUNCTIONAL PROGRAMMING?

- Formal Provability: Line-by-line invariants
- Modularity: Encourages small independent functions.

  $\Rightarrow$ Since the function does not depend on any external variable or state, calling it from a different piece of code is straightforward.
- Composability: Arrange existing functions for new goals
- Easy Debugging Behavior: Depends only on input
- Brevity: Functional programming is often less verbose than other paradigms.

Lambda Functions

LAMBDA FUNCTIONS

- Anonymous, on-the-fly, unnamed functions

```
# Keyword lambda creates an anonymous function
# Returns an expression
lambda params: expr(params)
```

Defined Functions vs. Lambdas

```
# def binds a name to a function object
def greet():
    print("Hi!")

# lambda only creates a function object
lambda val: val ** 2
lambda x, y: x * y
lambda pair: pair[0] * pair[1]

(lambda x: x > 3)(4) # => True
# Creates a function object and immediately call it
```

USING LAMBDAS

```
# Squares from 0**2 to 9**2
map(lambda val: val ** 2, range(10))

# Tuples with positive second elements
filter(lambda pair: pair[1] > 0, [(4,1), (3, -2), (8,0)])

# Sort a collection based on a custom function.
sorted([(4,1), (3, -2), (8,0)], key=lambda pair: pair[1])

triple = lambda x: x * 3 # This is bad. Why?
```

- Why use lambdas?
  - Avoids defining lots of small, one-use, simple functions
  - Declutters your local namespace
  - Presents function implementation inline at the call site