Functions in Python
Part 1

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Terminology
  - return values
  - calls and passing parameters
- Function as object
- Variable-length arguments
  - by position (tuple), by keywords (dictionary)
  - Unpacking arguments by *tuple and **dict

Function

- a *callable* object
  - pass parameters to it, if any
  - get a return value, or None

```
>>> def hello():
...     print('hello world')
...
>>> res = hello()
hello world
>>> print(res)
None
```

Terminology

```
>>> def Double(x):
...     return 2 * x
...
>>> Double(23)
46
>>> Double(27 - 5 + 13)
70
```

- function name:
  - Double
- formal parameter:
  - x
- actual parameter:
  - 23 (when you call Double(23))
  - 35 (when you call Double(27-5+13) which is Double(35)
  - i.e., expressions get evaluated before being passed as params
- "argument", "parameters" are interchangeable

Return value

- can be any type
  - can even be another function as an object!
  - can return multiple objects -- as a tuple
  - returns `None` by default if you don't do anything
- Example: prompt user for name & passwd

```
def getNamePassword():
    userName = input('Username: ')
    passwd = input('Password: ')
    return userName, passwd # this is a tuple!
    # same as (userName, passwd)
```

- caller can do unpacking assignment
  - `name, passwd = getNamePassword()`

Function object vs. function call

- *functionName* references function object
- *functonName*( ) is a function call

```
>>> getNamePassword   # this references the function object
<function getNamePassword at 0x10ca9bb70>
>>> getNamePassword() # the () makes the call
Username: Mary
Password: Little Lamb
('Mary', 'Little Lamb')
```

- Both are expressions
  - functionName is a "callable object" (code)
  - function call evaluates to the return value of the call

function assignment

- you can assign a function object to another name
- Be careful!! You can redefine a function name to anther value!

```
>>> p = print  # print is a built-in function
>>> p('hello world')  # this calls the built-in print function
hello world
>>> print = getNamePassword  # print is like a variable
>>> print()  # this doesn't call print; calls getNamePassword()!!
Username: Mary
Password: Little Lamb
('Mary', 'Little Lamb')
>>> print('hello world') # this doesn't call the built-in print()!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: getNamePassword() takes 0 positional arguments but 1
was given
```

two ways of passing parameters: *by position* and *by name*

- Example: built-in function `open`(*file*, *mode*)
- By position: ordering must match
  - `fh = open('myfile.txt', 'r')`
    i.e., `'myfile.txt'` for name, `'r'` for mode
- By name: explicitly name the formal params
  - `fh = open(file = 'myfile.txt', mode='r')`
  - `fh = open(mode='r', file='myfile.txt')`
    `# same thing: ordering doesn't matter!!`

Why pass parameters *by name*?

- Better readability
  - "self-documenting" code
  - be more specific, say what you mean
- Less likely to make mistakes
  - some functions may have a long list of parameters
  - keywords help programmers match up actual params with formal params
- Especially useful for functions with optional parameters (i.e., with default values)

Optional Parameters with Default Values

- Example: open(file, mode)
  - the mode defaults to `'r'` (reading)
  - can override the default value
- Declaration:
  - assign default value to the formal parameter

```
>>> def withTax(price, rate=0.05):
...     return price * (1 + rate)
...
>>> withTax(100)
105.0
>>> withTax(200, 0.07)
214
```

Restrictions on ordering of optional parameters

- Optional parameters must be declared after the required parameters
- Good:

```
def withTax(price, rate=0.05):
```

- Not good

```
def withTax(rate=0.05, price):
```

  - because you can't say `withTax( , 100)` just to use the default rate

Evaluation of default parameter

- the default value is evaluated **once for all**

```
import random
def withTax(price, rate=random.randint(1,100)/100.):
    return price * (1 + rate)
```

- the default tax rate is randomly chosen **at the time the function is defined**
    - it does not generate a new random tax rate each time you call it!
- This also means default expression **cannot involve another parameter**
    - e.g., `def F(a, b=a)` is bad!

Variable-length argument list

- want to pass a variable number of arguments to a call
  - they don't necessarily have a default parameter
- example: total(), max(), average(), ...
  - don't want to necessarily pass a list or tuple
- example: suppose we want a function

$$\texttt{totalTax(}\underbrace{\texttt{0.05}}_{\text{tax rate}}\texttt{, }\underbrace{\texttt{10, 20, 23, 18}}_{\text{price list}}\texttt{)}$$

Declaring variable-length arguments

- Put a * in front of the var-length argument
  - They appear to the function as a **tuple**
  - The caller pass them as variable-length arguments

```
def totalTax(rate, *priceArgs):
    sum = 0
    for i in priceArgs:  # iterate over elements of tuple
        sum = sum + i
    return(1 + rate) * sum
```

- Call it as a flat list, not as a tuple

```
totalTax(0.05, 10, 20, 23, 18)
```
✓

```
totalTax(0.05, (10, 20, 23, 18))
```
⛔

passing variable-keyword arguments by name

- conceptually like a dictionary, but flatten
- example:

```
def totalTaxByItem(rate, priceDict):
```

  - if implemented as a dictionary, call it as

```
totalTaxByItem(rate=0.05, priceDict={'eggs': 10, 'soap': 20,
'bread': 25})
```

    - but it is awkward to pass an inline dictionary
    - want name=value style as in function calls, rather than key:value style as in dictionary

```
totalTaxByItem(rate=0.05, eggs = 10, soap = 20, bread = 25)
```

these names are chosen by the caller, not the function itself!!

Declaring variable-keyword parameter list

- Put ** in front of the variable-keyword formal parameter

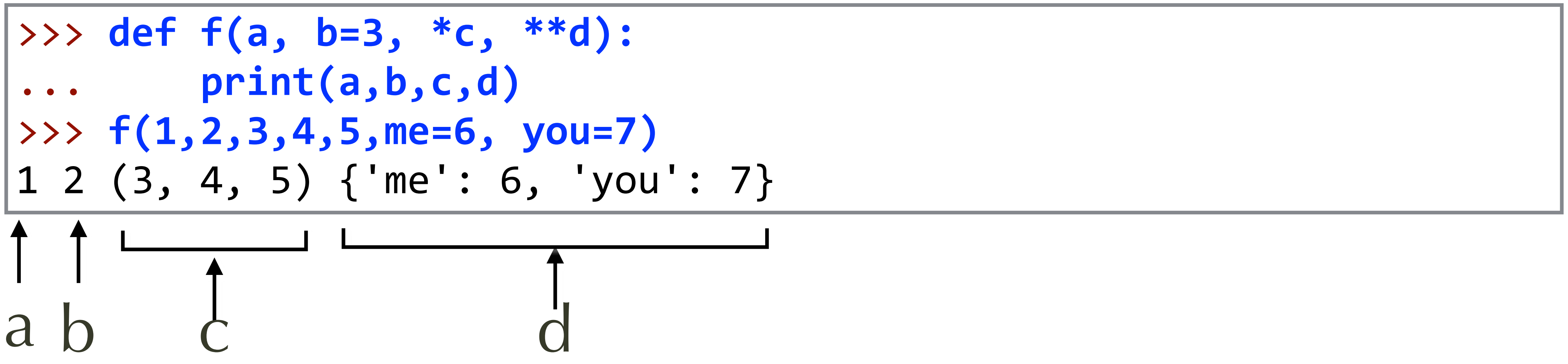
```
def totalTax(rate, **priceDict):
    sum = 0
    for i in priceDict.values():
        sum = sum + i
    return(1 + rate) * sum
```

  - in this case, the caller can use its own name=value words; we don't care what words they pass

- In general, the function may need to examine the keys and give special interpretation

Combining default value, variable length, and variable keyword arguments

- Order:
  - default value first
  - followed by variable-length tuple
  - followed by variable-keyword dict
- Can check this with a simple example



```
>>> def f(a, b=3, *c, **d):
...     print(a,b,c,d)
>>> f(1,2,3,4,5,me=6, you=7)
1 2 (3, 4, 5) {'me': 6, 'you': 7}
```

* Unpacking a sequence for parameters

- want to print 0 1 2 3 4 5 6 7 8 9
  - but want to use range(10) or list to generate

```
>>> L = range(10)
>>> print(range(10))  # first attempt
range(0, 10)
>>> print(list(L))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Solution: * unpacking
  - unpacks a sequence into separate actual params

```
>>> print(*L)   # effectively print(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
0 1 2 3 4 5 6 7 8 9
```

Example of unpacking parameters from sequence

- try out different #arguments to `range()`

```
>>> L = [(1, ), (1, 6), (1, 9, 2), (10, 0, -2)]
>>> for R in L:
...     print(list(range(*R)))
...
[0]
[1, 2, 3, 4, 5]
[1, 3, 5, 7]
[10, 8, 6, 4, 2]
```

- would not work without the *

```
>>> for R in L:
...     print(list(range(R)))
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'tuple' object cannot be interpreted as an integer
```

Unpacking multiple sequences as parameters to a call

- OK to use several * unpacking expressions within a call
  - they just get joined together as parameter list

```
>>> A = (1, 2)
>>> B = (3, 4, 5)
>>> max(A, B)     # max((1, 2), (3, 4, 5))
(3, 4, 5)
>>> max(*A, *B)   # max(1, 2, 3, 4, 5)
5
```

** Unpacking dictionary as named parameters

- Put ** in front of dictionary to unpack the key-value pairs as name=value parameters

```
>>> D = {'file': 'arg.py', 'mode': 'r' }
>>> fh = open(**D)
>>> data = fh.read()
>>> fh.close()
```

- `fh = open(**D)` in this case is the same as
  `fh = open(file='arg.py', mode='r')`

Summary

- Function definition
  - formal parameters
  - default value, variable-length, variable-keyword
  - return type
- Function invocation (call)
  - actual parameters, positional vs. named
  - unpacking sequence for positional parameter
  - unpacking dictionary for named parameters