Lecture #12: Misc #1

Presented by Jamal Bouajjaj

2023–10–02

For University of New Haven's Fall 2023 CSCIxx51 Course

Misc

This lecture/presentation is for a collection of stuff that I either missed, or stuff which is small to not warrant a whole class for them individually, but enough for multiple of them

Tuple ”Collapsing”

Tuple Collapsing

If you declare a tuple with one element, by default it will ignore your parenthesis, making it into a not-tuple as parenthesis can also be used to have order of operation.

To make a one-element tuple into one, add a comma after the first element

```
print(type( (1) ))   # int
print(type( (1,) ))  # tuple
```

Unpacking and Packing

List Unpacking

What if you have a Python list, where that list is actually input arguments to a function. Can you pass all of them to the function?

List Unpacking

What if you have a Python list, where that list is actually input arguments to a function. Can you pass all of them to the function?

YES, by unpacking the list. Unpacking just means to take the elements of the sequence, and set them as arguments to a function. This is done with a star *. For example:

```
def addThreeNumbers(a, b, c):
    return a + b + c
# The following are equivalent
addThreeNumbers(1, 2, 3)
addThreeNumbers(*[1, 2, 3])
```

Dict Unpacking

If done with a dictionary, then the keys become the variable identifier, and the value is...well...the value of that identifier. Dictionary unpacking is done with two stars **.

```
def addThreeNumbers(a, b, c):
    return a + b + c
# The following are equivalent
addThreeNumbers(a=1, b=2, c=3)
addThreeNumbers(**{'a': 1, 'b': 2, 'c': 3})
```

Function Packing

Now what if you need a function to take multiple user inputs, but have the flexiblity to have any amount. You CAN have the user just enter a list as an argument, but there is another way.

Function Packing

Now what if you need a function to take multiple user inputs, but have the flexiblity to have any amount. You CAN have the user just enter a list as an argument, but there is another way.

The same syntax for list unpacking sort of works backwards if it's an argument of a function: Take all of the keywords by the user, and pack them into a list. The text *args* isn't fixed, but it's the standard

```
def addNumbers(*args):
    return sum(args)
addNumbers(1, 2, 3)
addNumbers(1, 2, 3, 4, 5, 6, 7)
```

Function Dict Packing

Same concept works for dictionaries. The standard text is *kwargs*

```
def addNumbers(**kwargs):
    if 'a' in kwargs:
        return 'A in args'
    return 'no a in args'
addNumbers(a=2, b=1, fes=1)
addNumbers(hb=3, bc=1, bjh=2)
```

Why not both?

```
def addNumbers(*args, **kwargs):
    print(len(args))
    if 'a' in kwargs:
        return 'A in args'
    return 'no a in args'

addNumbers(2, 4, 1)
addNumbers(2, 4, 1, a=2, b=1, fes=1)
addNumbers(hb=3, bc=1, bjh=2)
```

Lambda

Lambda Function

A lambda function is constructor to make anonymous functions. What lambda returns is a function that can be called.

An anonymous function (a general programming term) is a function without a name.

Lambda Function

A lambda function is constructor to make anonymous functions. What lambda returns is a function that can be called.

An anonymous function (a general programming term) is a function without a name.

A λ function has the following syntax:

```
lambda *VAR: SOMETHING
```

The function above takes arguments, and whatever that SOMETHING does is what the lambda function returns.

Example

```
def stripFunction(a):
    return a.strip()
with open('file') as f:
    r = map(stripFunction, f)
    # OR
    r = map(lambda x: x.strip(), f)
```

Other use

Also useful for having a function that will execute later on with arguments:

```
def ret(x):
    print(x)

functions = []
for i in range(100):
    #functions.append(ret(i))   # <- not good!
    functions.append(lambda i=i: ret(i))

functions[0]()    # a bit cursed...no?
```

Block Documentation

Block

If you need multiline documentation, you need need individual #. OR, you can just surround your comment in 3x”

```
"""
This is a block documentation

Anything in here is a comment
"""

"""Another valid block doc: don't have to be
↪  multi-line"""
```

Reservations

With that said, the convention is to keep block documentation only for function, class, or module docs. These are called *docstrings*

```
def your_function(a, b):
  """
  This functoin returns if a > b

  Args:
    a: number
    b: number
  """
  return a > b
```

Formats

There are multiple docstring format conventions if you want to follow them. They tend to include all of a function's info like arguments, return type, exceptions, etc. They can also be used to automatically generate a documentation webpage.

See `https://stackoverflow.com/questions/3898572/what-are-the-most-common-python-docstring-formats` for the different formats.

PyCharm can handle all of them, and will auto-fill a docstring for a function when you make one.

Getting docstrings

You can also have Python return a docstring of a function or class by calling `__doc__`:

```
import random
print(random.randint.__doc__)
```

Type Hinting

Hinting

If you have a function as follows, let's say it's expecting a certain input type

```
def really_cool_function(money):
  if money < 50:
    print("You broke")
  else:
    print("You not broke")
```

And this will fail if the type isn't a float or integer. How do you convey it?

Docs

You could have the block documentation state so

```
def really_cool_function(money):
  """
  Really cool function

  Args:
    money (int): This is an integer!
  """
  if money < 50:
    print("You broke")
  else:
    print("You not broke")
```

But what if there was a BETTER way, one in which the IDE can also understand and do static checking to that who uses this function?

Hinting

Introducing type hinting! Now with this one small trick (colon), you can have the user, IDE, and any static checker know what type your function is expecting!

After a variable, you type colon with the type class it expects

```
def really_cool_function(money: int):
  if money < 50:
    print("You broke")
  else:
    print("You not broke")
```

Hinting 2

Also works with optional variables, before the equate sign.

```
def really_cool_function2(money: int, areyoucool: bool =
↪  False):
  if areyoucool:
    print("You're cool anyways, who needs money!")
    return
  if money < 50:
    print("You broke")
  else:
    print("You not broke")
```

Global

Variable Scope

Variables have a "scope" to them, i.e what part of code they encompass.

For example, a variable declared in a function has it's scope withing the function, and cannot be accessed externally:

```
def really_cool_function3():
    money = 5

really_cool_function3()
print(money)    # This will fail
```

Variable Scope

But a function can access outside scope variables if it is not defined

```
def really_cool_function4():
    print(cash)

cash = 50
really_cool_function4()
```

Variable Scope

If function uses a local variable, even if later, then the global variable is not used:

```
def really_cool_function4():
    print(cash) # this will fail
    cash = 10

cash = 50
really_cool_function4()
```

Global

Unless you have a `global` keyword to have the scope of the variable be outside of the function. This will also allow functions to change global variables

```
def really_cool_function4():
    global cash
    print(cash)
    cash = 10
    print(cash)

cash = 50
print(cash)
really_cool_function4()
print(cash)
```

Guidance

My advice: *AVOID globals if possible*.

They can lead to unexpected bugs, and aren't a good design practice. For larger/robust applications, each function should have a defined input and output. If some variables needs to be kept in a state, use classes.

End

The end