Chapter 03 - Functions

May 8, 2021

0.1 Item 19: Never Unpack More Than Three Variables When Functions Return Multiple Values

- you can return multiple values as tuples in python
- you can use `Catch-All` Unpacking for function return

- using a large number of variables is error prone, espically if they are numerical
- this could be three individual values or 2 individual and one starred
- if you need to unpack more than three, your better off using a lightweight `class` or `namedtuple`

0.2 Item 20: Perfer Raising Exceptions to Returning `None`

[1]:
```
def careful_divide(a: float, b: float) -> float:
    '''
    Divides a by b.

    Raises:
        ValueError: When the inputs cannot be divided
    '''
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs')
```

0.3 Item 21: Know How Closures Interact with Variable Scope

- python supports **closure**: meaning functions refer to variables from the scope in which they were defined
- Functions are `first-class` objects in python, meaning you can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and `if` statements
- this is why you can pass closure functions as the `key` argument

- when you refrence a variable in an expression, python interpreter traverses the scope to resolve the reference in order
  1. the current functions scope
  2. any enclosing scopes (such as other containing functions)
  3. the scope of the module that contains the code (also called the `global scope`)
  4. the built-in scope (that contains functions like `len` and `str`)

- if none of these places are defined a variable with the referenced name, then a `NameError` exception is raised

- in the code below, `found` will never change to true because `nonlocal` is not used

[3]:
```python
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True # Seems simple
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

- python treats closure variables as new variables even if they have the same name as a global variable
- in Python, there is a special syntax for getting data out of a closure, and as mentioned it is `nonlocal`
- the limit on `nonlocal` is it wont traverse up the module-level scope
- dont use `nonlocal` for anything but simple function

- when `nonlocal` starts getting complicated, its better to wrap your state in a helper class

[6]:
```python
class Sorter:
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)

numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

0.4 Item 22: Reduce Visual Nosie with Variable Positional Arguments

- accepting a variable number of positional arguments can reduce noise `*args`
- if you pass a `list` using `*`, then it will pass each element in the list as a seprate element

[9]:
```python
def log(message, *values):
    if not values:
```

```
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{message}: {values_str}')

log('My numbers are', 1, 2)
log('Hi there') # Much better
```

```
My numbers are: 1, 2
Hi there
```

- the **first issue** with accpeting a variable number of possible arguments is that these optional positional arguments are always turned into a `tuple` before they are passed to a function
- this means that if the caller of a function uses the `*` operator on a generator, it will be iterated untill it's exhausted
- the resulting tuple includes every value from the generator, which could consume alot of memory and cause the program to crash

[18]:
```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)
```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

- functions that accept `*args` are best for situations where you know the number of inputs in the argument list will be reasonably small
- the **second** issue with `*args` is that you cant add new positional arguments to a function in the future without migrating every caller

0.5 Item 23: Provide Optional Behavior with Keyword Arguments

[23]:
```
def remainder(number, divisor):
    return number % divisor
```

- if you already have a dictionary and you want to use its contents to call a function like `remainder` you can do this by using the `**` operator

[29]:
```
my_kwargs = {
    'number': 20,
    'divisor': 7
}

assert remainder(**my_kwargs) == 6
```

- you can mix and match the operator or use multiple `**kwargs`

[33]:
```python
my_kwargs = {
 'divisor': 7,
}

assert remainder(number=20, **my_kwargs) == 6

my_kwargs = {
 'number': 20,
}

other_kwargs = {
 'divisor': 7,
}
assert remainder(**my_kwargs, **other_kwargs) == 6
```

- first benefit is that keyword arguments make the function call clearer to new readers of the code
  - with the `keyword` arguments, `number=20` and `divisor=7` make it immediately obvious which parameter is being used for each purpose
- the second benefit of keyword arguments is that they can have default values specified in the function definition and thus eliminates repetitive code
- the thid reason to use keyword arguments is that they provide a powerful way to extend a functions parameters while remaining backword compatible with existing callers
  - this means you can provide additional functionality without having to migrate alot of existing code
- optional keyword arguments should always be passed by keyword instead of by position

0.6 Item 24: Use `None` and Docstrings to Specify Dynamic Default Arguments

- sometimes you need to use a non-static type as a keyword argument's default value
- for example if I want to print logging messages that are marked with the time of the logged event
- in the default case, I want the message to include the time when then function was called
- I might use the following approach

[35]:
```python
from time import sleep
from datetime import datetime

def log(message, when=datetime.now()):
    print(f'{when}: {message}')

log('Hi there!')
sleep(0.1)
log('Hello again!')
```

```
2021-04-12 08:37:12.308404: Hi there!
2021-04-12 08:37:12.308404: Hello again!
```

- the code above does not work as intended because the time stamps are the same
- that is because `datetime.now` is executed only a single time, when the function is defined
- a default argument value is evaluated only once per module load
- after the module containing the code is loaded, the `datetime.now()` default argument will never be evaluated again

- in python we address this by providing a default value of `None` and to the document the actual behavior

[37]:
```python
def log(message, when=None):
    """
    Log a message with a timestamp

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')

log('Hi there!')
sleep(0.1)
log('Hello again!')
```

```
2021-04-12 08:41:49.145273: Hi there!
2021-04-12 08:41:49.249204: Hello again!
```

- using the `None` default aregument is espically important when the arguments are mutable
- for example, say I want to load a value encoded as `JSON` data
- if decoding the data fails, I want an empty dictionary to be returned

[41]:
```python
import json

def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default

foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
```

```
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

- the problem above is the same as the `datetime.now` example
- the dectionary specified for `default` will be shared by all calls to `decode` because `default` argument values are evaluated only one
- you might have expected two different dictionaries, but modifying one modifes the other one

[42]:
```
def decode(data, default=None):
    """
    Load JSON data from a string

    Args:
        data: JSON data to decode
        default: Value to return if decoding fails
            Defaults to an empty dictionary
    """
    try:
        return json.loads(data)
    except ValueError:
        if default is None:
            default = {}
        return default

foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
assert foo is not bar
```

```
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

[43]:
```
from typing import Optional

def log_typed(message: str, when: Optional[datetime]=None) -> None:
    """ Log a message with a timestamp

    Args:
        message: Message to print
        when: datetime of when the message occurred.
            Defaults to the present time
    """
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')
```

0.7 Item 25: Enforce Clarity with Keyword-Only and Postional-Only Arguments

- when you have complex behavior in a function and the the caller can use any parameters, it might be a good idea to enforce things
- we can redefine a function to accept keyword-only arguments
- the `*` symbol in the argument list indicates the end of positional arguments and the beginning of keyword-onyl arguments

[45]:
```
def safe_division(number, divisor, *,
                        ignore_overflow,
                        ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise

# this will no longer work
# safe_division(1.0, 10*500, True, False)

# this will work
safe_division(1.0, 0, ignore_zero_division=True)
```

- but even that is not good because callers may specify the first two required arguments (`number` and `divisor`) with a mix of postions and keywords
- python offers a solution called `postional-only` arguments
- these arguments can be supplied only by position and never by keyword (oppiste of the keyword only example above)

- the symbol `/` in the argument lsit indicates where positional-only arguments end

[49]:
```
def safe_division_d(numerator, denominator, /, *, # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    pass
```

- in the above example, you cannot do this
  - `safe_division_d(numerator=2, denominator=5)`
- but you have to do this
  - `ignore_overflow=False, ignore_zero_division=False`

- a problem still exists where between the `/` and the `*`, you can arguments may be passed either by position or keyword

0.8 Item 26: Define Function Decorators with `functools.wrap`

- a decorator has the baility to run additional code before and after each call to a function it wraps

[53]:
```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(f'{func.__name__}({args!r}, {kwargs!r}) '
              f'-> {result!r}')
        return result
    return wrapper

@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n-2) + fibonacci(n-1))
```

- using the `@` symbol is equivalent to calling the decorator on the function it wraps and assigning the return value to the orginal name in the same scope
- the decorator function runs the wrapper code before and after fibonacci runs
- it prings the arguments anf return value at each level in the recursive stack

- the one issue with a wrapper is that if you use `print(fibonacci)` you dont get `fibonacci` but rather `trace`
- the `trace` function returns the wrapper defined within its body
- the wrapper function is whats assigned to the `fibonacci` name in the containing module because of the decorator
- object serializers brake because they cant determine the location of the orginal fucntion that was decorated

- the solution is to use the `wraps` helper from the `functools` module
- this is a decorator that helps you write decorators
- when you apply it, it copies all of the important metadata about the inner function to the outter function

[54]:
```
from functools import wraps

def trace(func):
    @wraps(func)
    def wrapper(args, *kwargs):
        pass
    return wrapper

@trace
```

```python
def fibonacci(n):
    pass
```

- Use the wraps decorator from the functools built-in module when you define your own decorators to avoid issues