CS61A Lab 3: Higher-Order Functions

Week 3, 2011

Exercise 1: Type-checking

Many procedures require a certain type of argument. For example, many arithmetic procedures only work if given numeric arguments. If given a non-number, an error results. For instance, `pow('hello', 2)` will result in the following error:

```
TypeError: unsupported operand type(s) for ** or pow(): 'str' and
'int'
```

Suppose we want to write safe versions of procedures, that can check if the argument is okay, and either call the underlying procedure or return `False` for a bad argument instead of giving an error. (We'll restrict our attention to procedures that take a single argument.)

```
>>> sqrt('hello')
Traceback (most recent call last):
 File "", line 1, in
TypeError: a float is required
>>> type-check(sqrt, isnumber, 'hello')
False
>>> type-check(sqrt, isnumber, 4)
2
```

1.) Write `type-check`. Its arguments are a function, a type-checking predicate that returns `True` if and only if the datum is a legal argument to the function, and the datum.
For testing, you'll want the `isnumber` procedure: use the one here (you don't have to understand how it works):

```
def isnumber(thing):
    try:
        int(thing)
    except:
        return False
    return True
```

2.) We really don't want to have to use `type_check` explicitly every time. Instead, we'd like to be able to use a `safe_sqrt` procedure:

```
>>> safe_sqrt('hello')
False
>>> safe_sqrt(4)
2
```

Don't write `safe_sqrt`! Instead, write a procedure `make_safe` that you can use this way:

```
>>> safe_sqrt = make_safe(sqrt, isnumber)
```

It should take two arguments, a function and a type-checking predicate, and return a new function that returns `False` if its argument doesn't satisfy the predicate.

Documentation Strings (aka docstrings)

From the PEP 257:

> "A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition."

A docstring can (and should!) be used to document functions that you write. Good docstrings will do things like:

- Concisely (and accurately) describe what the function does
- Describe input arguments and return values
- List any assumptions that the code makes (such as types of arguments)

Ideally, a docstring should be of such quality that a programmer will, after having read the docstring, know exactly what the function does, and how to use it. For instance, here's an example of using docstrings:

```
def square(n):
    """Given a number n, returns the result of squaring n.

    Arguments:
         n -- a number
    Returns:
         The square of n.
```

```
    """
    return n * n
```

Doctests

A really cool feature is the ability to create 'automated' test cases within the docstrings. Inside a docstring, you can signify expected behavior by typing out expected interpreter behavior, like:

```
def square(n):
    """Given a number n, returns the result of squaring n.

    Arguments:
        n -- a number
    Returns:
        The square of n.

    >>> square(5)
    25
    >>> square(-1)
    1
    """
    return n * n
```

To 'run' the test cases, go to the terminal and run `python` with the '`-m doctest -v`' options:

```
star [123] ~/lab3 # python -m doctest -v square.py
Trying:
    square(5)
Expecting:
    25
ok
Trying:
    square(-1)
Expecting:
    1
ok
1 items had no tests:
    doctest_test
1 items passed all tests:
  2 tests in doctest_test.square
2 tests in 2 items.
2 passed and 0 failed.
```

```
Test passed.
```

The '`-m doctest`' option tells Python to run all doctests in `square.py`, and the '`-v`' option tells Python to tell us everything that's going on with each doctest (i.e. 'verbose' mode). If you only care about doctests that fail, don't provide the '`-v`' option. If a doctest fails, then Python will let you know which doctest failed.

As you can imagine, writing doctests is a quick and convenient way to both provide easy-to-verify test cases, and also provides example usage inside the docstring.

3.) Add doc-tests to your make_safe procedure.

Project 1 Peek

This section is a sort of sneak peak of project 1. In project 1, you'll be dealing with randomness (in the form of dice rolls). Often when dealing with non-deterministic procedures, we want to be able to quantify how a non-deterministic procedure typically behaves. One common quantification is called the mean, aka the average.

For the following exercises, we'll be using a simulated die for our source of randomness. Copy the file `dice.py` into your current directory (say, `lab3/`).

Look through `dice.py` - there are two functions, `make_fair_die` and `make_test_die`.

make_fair_die

This procedure takes in an integer `sides`, and returns a procedure that, when invoked, simulates a die roll by returning a number from `1` to `sides`. For instance, to create a 6-sided die, we would do:

```
>>> from dice import *
>>> my_die = make_fair_die(6)
>>> my_die()
4
>>> my_die()
2
```

> Note: It turns out that providing an integer argument to `make_fair_die` is optional - if you look at the code for `make_fair_die` in `dice.py`, the function signature looks like:

```
def make_fair_die(sides=6):
    …
```

The `sides=6` means that the `sides` argument is optional - if the caller doesn't provide an argument, then `make_fair_die` will set `sides` to the default value specified (in this case, `6`).

```
>>> my_die = make_fair_die()     # Also creates a 6-sided die
>>> my_die()
5
```

make_test_die

This procedure lets you explicitly state exactly how the die will behave. For instance, if I want to create a die that will always return `2`, `5`, `3` over and over again, I would do:

```
>>> die = make_test_die(2, 5, 3)
>>> die()
2
>>> die()
5
>>> die()
3
>>> die()
2
```

Note: The above interaction may seem a little morbid - remember, `die` is the singular version of dice. An unfortunate coincidence.

3.) Implement the `average_value` function (Q8 from Project 1). This function takes two arguments: a non-pure target function `fn` and an integer `num_samples` that indicates how many times the target function should be called. The return value of the target function should be a number. Call `fn num_sample` times, and return the average result. Assume that `fn` takes no arguments. Make sure that the doctest passes before you move on.

```
from ucb import main, trace, interact

def average_value(fn, num_samples):
    """Compute the average value returned by fn over num_samples trials.
```

```
    >>> d = make_test_die(1, 3, 5, 7)
    >>> average_value(d, 100)
    4.0
    """
    "*** YOUR CODE HERE ***"
```

4.) Now write the `averaged1` function. `averaged1` should take two arguments: a no-argument function `fn`, and an integer `num_samples`.
`averaged1` should return a new no-argument function that, when called, returns the average value of calling `fn` `num_samples` number of times.

```
def averaged1(fn, num_samples=100):
    """Return a function that returns the average_value of fn when called.

    >>> die = make_test_die(3, 1, 5, 7)
    >>> avg_die = averaged1(die)
    >>> avg_die()
    4.0
    """
    "*** YOUR CODE HERE ***"
```

Assert statements

As your code becomes more and more complicated, it becomes harder to keep track of things (the curse of complexity!). If left unchecked, the smallest bug can lead to hours of painful debugging.

One Python feature aimed at making the debugging process easier is the `assert` statement.

`assert` is a statement that is used in the following way:

```
>>> assert 1 == 2
Traceback (most recent call last):
 File "", line 1, in
AssertionError
>>> assert 1 == 1
>>> assert 42
```

```
>>>
```

So, `assert` takes in an expression, and if the expression evaluates to a true value, then execution continues as normal. If the expression evaluates to a false value, then an `AssertionError` is raised.

You can also add a helpful message to be displayed when the assertion fails:

```
>>> assert 2 + 2 == 5, "Only for sufficiently large values of 2."
Traceback (most recent call last):
 File "", line 1, in
AssertionError: Only for sufficiently large values of 2.
```

Testing

It's a good idea to get in the habit of writing tests for your own code. The idea is that if you can gain confidence in the small pieces of your program (i.e. each individual function), then it's easy to combine each piece to create a more complex program. The way to gain confidence is by passing the test cases that you write.

For instance, say we had a function that repeated an input string some number of times:

```
def repeat_word(str, n):
    result = str
    count = 0
    while count < num:
         result = result + str
         count += 1
    return result
```

An example of a test suite would be something like:

```
def test_repeat_word():
    assert repeat_word('hi', 0) == 'hi'
    assert repeat_word('hi', 1) == 'hihi'
    assert repeat_word('', 1) == ''
    assert repeat_word('hello', 3) == 'hellohellohello'
```

Good testing strategies include doing the following:

- Test the 'edge' cases, like passing in the empty string `""`, or passing in `0`.

- Test the 'general' case.

In fact, there's a style of programming called Test-Driven Development (TDD), where you write the test cases first, and then you write the code. The idea is that, writing the test cases first accomplishes several goals. First off, you have a solid test suite to run against from the beginning. Second off, writing the test cases (i.e. expected input/output behavior) tends to fully flesh out the exact behavior of the function in your head, which will help when you go to actually write it.

5.) Think out how you can test your `averaged` and `averaged1` functions. It's not as straightforward as testing something like `repeat_word`, since `repeat_word` is a deterministic function, whereas `averaged`/`averaged1` will return slightly different values for the same input (assuming you're using some sort of random function as part of the input).

Then, write a set of good test cases.