Strings and Tuples

CSE/IT 107L

NMT Department of Computer Science and Engineering

"All thought is a kind of computation."

— D. Hobbes

"It [programming] is the only job I can think of where I get to be both an engineer and artist. There's an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation."

— A. Hertzfeld

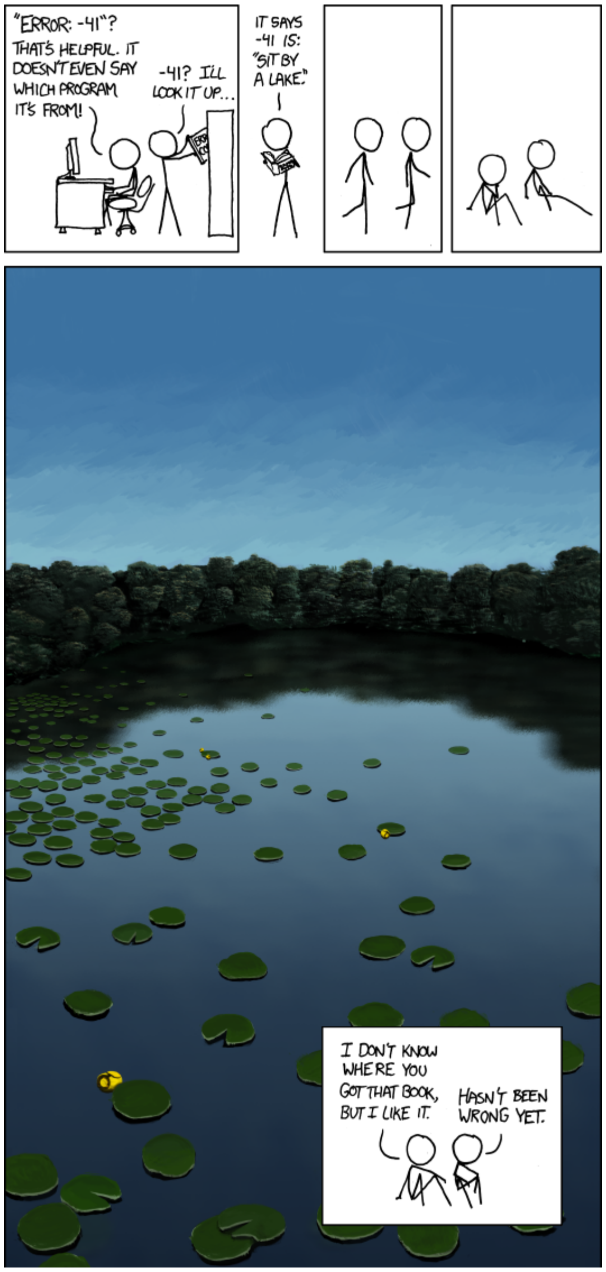
"ERROR: -41"? THAT'S HELPFUL. IT DOESN'T EVEN SAY WHICH PROGRAM IT'S FROM!
-41? I'LL LOOK IT UP...
IT SAYS -41 IS: "SIT BY A LAKE."
I DON'T KNOW WHERE YOU GOT THAT BOOK, BUT I LIKE IT.
HASN'T BEEN WRONG YET.

Figure 1: `http://xkcd.com/1024`

Introduction

This lab will build on your understanding of strings and introduce a new tool: tupples. You will learn how to implement and apply tupples to your programs. Like strings, tuples support indexing, slicing, and other similar features.

The lab also introduces keyword arguments, which are useful for multi-purpose functions or for those with arguments that have default values. With a more detailed knowledge of strings, you will be able to write documentation for functions in the form of docstrings, which will be visible using the `help()` function.

Contents

1 Tuples: Fixed-Length, Immutable Ordered Collections

Tuples work a lot like lists, except they cannot be modified; they are *immutable*. Instead of brackets `[]`, tuples are delimited by writing parentheses `()` around the elements. Tuples support slicing, just as lists do.

```
>>> food = ('eggs', 'bananas', 'lemonheads')
>>> food[1]
'bananas'
>>> food[1:3]
('bananas', 'lemonheads')
>>> food[1] = 'steak' # tuples are immutable!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Notice Python raises a `TypeError` with message "'tuple' object does not support item assignment" if the code attempts to assign to an item within a tuple. If a tuple contains a mutable value, such as a list, then that element can be modified without restriction.

We can nest tuples and lists interchangeably:

```
>>> points = [(1, 5), (3, 5), (0, 2), (1, 1),  # shift+enter for multi-line
...           (2, 1), (3, 1), (4, 2)]
>>> points[0]
(1, 5)
>>> fubar = ([1, 2, 3], [4, 5, (7, 8, 9)], (10, 11))
>>> fubar
([1, 2, 3], [4, 5, (7, 8, 9)], (10, 11))
```

Tuples, like lists, can also be empty. Use `len()` to find the length of a tuple:

```
>>> foo = ()
>>> len(foo)
0
```

We can also define a tuple by simply separating values by commas: Conversely, we can *unpack* tuples, which means assigning each element of a tuple to a variable. For example:

```
>>> food = 'trail mix', 'nothing'
>>> tylerfood, chrisfood = food
>>> # equivalent to, but shorter than: tylerfood = food[0]; chrisfood = food[1]
>>> tylerfood
'trail mix'
>>> chrisfood
'nothing'
```

Unpacking is useful for assigning a fixed number of return values from a function.

2 Documenting Your Functions and Modules With Docstrings

The `help()` function introduced in the previous labs prints the given function's documentation which is known as a docstring. You can write detailed documentation for your functions by placing a triple-quoted string immediately under the beginning of a function definition (after the `def f(...):`).

The docstring is a comment describing the use of a function: the documentation should include information about each argument, the return value, and the function's behavior.

Function comments must follow the Python Enhancement Proposal 257 (PEP 257), a document that describes Python docstring conventions. It is found at:

`https://www.python.org/dev/peps/pep-0257/`

The highlights of PEP 257 are listed below.

For short functions, the docstring may be a one-liner:

```python
def midpoint(a, b):
    """Find and return the midpoint of the given a and b."""
    return (a+b)/2
```

For larger functions or for a longer explanation, you must write a short summary followed by as much text as needed:

```python
def calculate_weekly_pay(pay_rate, hours, tax_rate):
    """Find net weekly pay after taxes with overtime set to 1.5 the pay rate.

    The net pay after taxes is calculated given the number of hours worked in a
    week, a pay rate, and a flat tax rate. Takes into consideration overtime pay
    at 1.5 times the pay rate.

    Arguments:
    pay_rate -- rate of pay
    hours -- number of hours worked in one week
    tax_rate -- flat tax rate (for example, 0.15 for 15%)
    """
    pay_before_taxes = hours * pay_rate

    # Add overtime payment if necessary
    if hours > 40:
        pay_before_taxes += (hours - 40) * pay_rate * 0.5

    pay_after_taxes = pay_before_taxes * (1 - tax_rate)
    return pay_after_taxes
```

Docstrings are Required

From now on, you will be required to put a *docstring* at the beginning of every function that you write.

3 Optional Function Arguments: Keyword Arguments

Functions may take zero or more required arguments, these are called *positional arguments*. There is a special syntax that can be used to define *optional arguments*. For example, the following function takes either one or two arguments. The second argument has a default value of 5.

```
>>> def f(fst, snd = 5):
...     return fst * snd
>>> f(10)
50
>>> f(10, 6)
60
>>> f()
Traceback (most recent call last):
  File ``<stdin>'', line 1, in <module>
TypeError: f() missing 1 required positional argument: 'fst'
```

An important feature is that keyword arguments may be labeled and listed in any order when calling the function. For example, the following function takes two keyword arguments. There are five valid ways to call this function. You may pass no arguments, then the default values are used. You may pass either argument by itself or you may pass both arguments in any order.

```
>>> def p(greeting='Hello', subject='world'):
...     return '{} {}'.format(greeting, subject)
>>> p()
'Hello world'
>>> p(greeting='Goodbye')
'Goodbye world'
>>> p(subject='humans')
'Hello humans'
>>> p(greeting='Bonjour', subject='le monde')
'Bonjour le monde'
>>> p(subject='le monde', greeting='Bonjour')
'Bonjour le monde'
```

4 Exercises

Exercise 4.1 (cipher.py).
A cipher is an algorithm that preforms encryption and decryption on some given information. These algoritms are used within programs to secure the data within the system. For this lab you will be constructing a simple reverse cipher that will encrypt and decrypt some given information.

The reverse cipher algorithm has the following features:

- Reverse ciphers use a pattern of reversing the string of plain text
- Encryption and decryption is treated as the same algorithm
- To decrypt the cipher, the algorithm used to encrypt the text is used

The major drawback to this encryption method is that it is very weak and easy to see through. This exercise is meant to give and introduction to what a cipher is and how to construct a simple one.

Build a program that takes in some plain text from the user and applies the reverse cipher to it. Make sure to use the boiler plate code and call the cipher function from the main function. The function should reverse the passed argument and print the resulting cipher to the console. You are not allowed to use the built in reversed() function that python provides and should work to solve the problem yourselves.

Sample output should look like:

```
Please enter in your message: This is an example
Your message is now: elpmaxe na si sihT
```

Exercise 4.2 (luhns.py).
Luhn's algorithm provides a quick way to check if a credit card is valid or not. The algorithm consists of four steps.

1. We will use the Diners Club card number 38520000023237 as an example.

$$3 \quad 8 \quad 5 \quad 2 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 2 \quad 3 \quad 2 \quad 3 \quad 7$$

2. Starting with the second to last digit (ten's column digit), multiply every other digit by two.

$$6 \quad 8 \quad 10 \quad 2 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 2 \quad 6 \quad 2 \quad 6 \quad 7$$

3. Sum all the digits of the resulting number. Note that for 10, you also sum its digits: $(1+0)$; for an 18, you would do $1+8$.

$$6+8+(1+0)+2+0+0+0+0+0+2+6+2+6+7=40$$

4. If the total sum modulo by 10 is zero, then the card is valid; otherwise it is invalid. For this example, the last step is to check if 40 modulus 10 is equal to 0, which is true. So the card is valid.

Write a program that implements Luhn's Algorithm for validating credit cards. It should ask the user to enter a credit card number and tell the user whether it is valid or not.

There must be a separate function called `validate` that takes in a card number and validates it. It should not print anything nor accept user input.

Testing Script

You can test your code in `luhns.py` by running the test script `test_luhns.py`. This will automatically test your function on various card numbers. The output of the test script may look like:

```
$ ls
luhns.py test_luhns.py ...
$ python3 test_luhns.py
luhns.validate("49927398717") should return False, but returned True.
luhns.validate("1234567812345678") should return False, but returned True.
The function luhns.validate correctly verified 3 out of 5 card numbers.
```

Exercise 4.3 (fractions.py).

Any fraction can be written as the division of two integers. You could express this in Python as a tuple — `(numerator, denominator)`.

For example, the fractions $\frac{1}{2}$, $\frac{10}{7}$, and $\frac{499}{10001}$ can be represented using the tuples `(1, 2)`, `(10, 7)`, and `(499, 10001)`.

Write the following functions:

`reduce(fraction)` This function takes a fraction, reduces it, and returns the result. For example, `reduce((8, 4))` should return `(2, 1)`. To reduce a fraction a/b, divide a and b by their GCD. The result is $(a/d)/(b/d)$. The math module comes with the `math.gcd` function.

`add(fraction1, fraction2)` Given two fractions as tuples, add them.

`multiply(fraction1, fraction2)` Given two fractions as tuples, multiply them.

`divide(fraction1, fraction2)` Given two fractions as tuples, divide them.

These functions should not use `input()` or `print()`.

Write a small command-line interface such that the user running your script sees something like this:

```
$ python3 fractions.py
Enter a fraction >>> 5/3
Enter a fraction >>> 10/3
Reduced fractions to 5/3 and 10/3.
Sum of the fractions: 3/1.
Multiplication of the fractions: 50/9.
Division of the first by the second: 1/2 .
$ python3 fractions.py
Enter a fraction >>> 3628800/479001600
Enter a fraction >>> 10/1000
Reduced fractions to 1/132 and 1/100
Sum of the fractions: 29/1650
Multiplication of the fractions: 1/13200
Division of the first by the second: 25/33
```

The `split` function may be useful when converting strings "xxx/yyy" into tuples (xxx, yyy). You should use the functions `add`, `multiply`, `reduce`, and `divide` from your main function.

Exercise 4.4 (README).
For the following labs you will be required to create aand submit a README file.

A README file is a simple text file that contains basic information regarding the purpose and use of the software program, utility, or game. README files often contain instructions or additional help regarding the software it accompanies.

For this lab you must have the following in your README file::

- **Purpose**: Describes what each program does and what problem it solves. You can keep this breif.
- **Conclusion**:
  - What you learned during the lab? What new aspect of programming did you learn from the lab? Be analytical about what you learned.
  - Did pair programming help in solving the problems and completing the prelab? Did you have problems with your budy?
  - Did you work with your budy on the lab? What sections did you discuss? Did you and your budy carry out a review session with each others' code?
  - Did you encounter any problems? How did you fix those problems?
  - What improvements could you make?

The conclusion does not have to be lengthy, but it should be thorough. You will include the README file with your submissions.

Index of New Functions and Methods

Submitting

You should submit your code as a `.zip` that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab5.zip`

Upload your .zip file to Canvas.

List of Files to Submit

Exercises start on page 4.