Lab 00 Getting Started

> The handout is adapted from cs61a 2020fall.

Environment Configuration

Before starting the lab, homework and project of sicp, you need to have an environment of `python3.8+`, `git`, `vscode`, `terminal(linux or unix like)`。

You need to check your environment like this:

```
$ git --version
git version 2.34.1 # any other version is ok
$ python3 --version
Python 3.9.12 # any version greater than 3.8 is ok
$ ls
# files and directories in your current working directory
```

> Note that if `ls` doesn't work in your terminal but `dir` does, please don't move on. Because you are using a windows comand prompt. Please switch to a unix-like(such as linux or mac-os) terminal.

You may need to grasp a few frequently used terminal commands:

- `ls`: **l**i**s**ts all files in the current directory
- `cd <path to directory>`: **c**hange into the specified **d**irectory
- `mkdir <directory name>`: **m**a**k**e a new **dir**ectory with the given name
- `mv <source path> <destination path>`: **mov**e the file at the given source to the given destination

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-lab00.git

```
git clone https://github.com/JacyCui/sicp-lab00.git
```

`lab00.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip lab00.zip
```

`README.md` is the handout for this lab. `solution` is a probrab solution the the lab. However, I might not give my solution exactly when the lab or homework is posted. You need to finish the task on your own first. If any problems occurs, please make use of the comment section.

> Please make a new directory for sicp, your directory tree may end up like this:

```
sicp
├── lab
│    └── lab00
│    ├── lab01
│    ├── lab01
│    └── ... ...
│
├── lectures
│    ├── lec01
│    ├── lec02
│    ├── lec03
│    └── ... ...
│
├── homework
│    └── hw01
│    ├── hw02
│    ├── hw03
│    └── ... ...
│
└── projects
  ├── proj01
  ├── proj02
  ├── proj03
  └── proj04
```

Python Basics

Expressions and statements

Programs are made up of expressions and statements. An *expression* is a piece of code that evaluates to some value and a *statement* is one or more lines of code that make something happen in a program.

When you enter a Python expression into the interactive Python interpreter, its value will be displayed. As you read through the following examples, try out some similar expressions on your own Python interpreter, which you can start up by typing this in your terminal:

```
python3
```

You'll be learning various types of expressions and statements in this course. For now, let's take a look at the ones you'll need to complete this lab.

Primitive expressions

Primitive expressions only take one step to evaluate. These include numbers and booleans, which just evaluate to themselves.

```
>>> 3
3
>>> 12.5
12.5
>>> True
True
```

Arithmetic expressions

Numbers may be combined with mathematical operators to form compound expressions. In addition to the `+` operator (addition), the `-` operator (subtraction), the `*` operator (multiplication) and the `**` operator (exponentiation), there are three division-like operators to remember:

- Floating point division (`/`): divides the first number by the second, evaluating to a number with a decimal point *even if the numbers divide evenly*.
- Floor division (`//`): divides the first number by the second and then rounds down, evaluating to an integer.
- Modulo (`%`): evaluates to the positive remainder left over from division.

Parentheses may be used to group subexpressions together; the entire expression is evaluated in PEMDAS order.

> Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right).

```
>>> 7 / 4
1.75
>>> (2 + 6) / 4
2.0
>>> 7 // 4        # Floor division (rounding down)
1
>>> 7 % 4         # Modulus (remainder of 7 // 4)
3
```

Assignment statements

An assignment statement consists of a name and an expression. It changes the state of the program by evaluating the expression to the right of the `=` sign and *binding* its value to the name on the left.

```
>>> a = (100 + 50) // 2
```

Now, if we evaluate `a`, the interpreter will display the value 75.

```
>>> a
75
```

Doing the assignment

Unlocking tests

One component of lab assignments is to predict how the Python interpreter will behave.

> Enter the following in your terminal to begin this section:
>
> ```
> python3 ok -q python-basics -u --local
> ```
>
> You will be prompted to enter the output of various statements/expressions. You must enter them correctly to move on, but there is no penalty for incorrect answers.
>
> Please pay attention of `--local` option, you must use the option since you are not a student of UC Berkeley, which means you can only test your code locally.

```
>>> 10 + 2
______

>>> 7 / 2
______

>>> 7 // 2
______

>>> 7 % 2          # 7 modulo 2, equivalent to the remainder of 7 // 2
______
```

```
>>> x = 20
>>> x + 2
______

>>> x
______

>>> y = 5
>>> y = y + 3
>>> y * 2
______

>>> y = y // 4
>>> y + x
______
```

Understanding problems

Labs will also consist of function writing problems. Open up `lab00.py` in your text editor. You can type `open .` on MacOS or `start .` on Windows to open the current directory in your Finder/File Explorer. Then double click or right click to open the file in your text editor. You should see something like this:

```
def twenty_twenty():
    """Come up with the most creative expression that evaluates to 2020,
    using only numbers and the +, *, and - operators.

    >>> twenty_twenty()
    2020
    """
    return ______
```

The lines in the triple-quotes `"""` are called a **docstring**, which is a description of what the function is supposed to do. When writing code in SICP, you should always read the docstring!

The lines that begin with `>>>` are called **doctests**. Recall that when using the Python interpreter, you write Python expressions next to `>>>` and the output is printed below that line. Doctests explain what the function does by showing actual Python code. It answers the question: "If we input this Python code, what should the expected output be?"

In `twenty_twenty`,

- The docstring tells you to "come up with the most creative expression that evaluates to 2020," but that you can only use numbers and arithmetic operators `+` (add), `*` (multiply), and `-` (subtract).
- The doctest checks that the function call `twenty_twenty()` should return the number 2020.

> You should not modify the docstring, unless you want to add your own tests! The only part of your assignments that you'll need to edit is the code.

Writing code

Once you understand what the question is asking, you're ready to start writing code! You should replace the underscores in `return ______` with an expression that evaluates to 2020. What's the most creative expression you can come up with?

> Don't forget to save your assignment after you edit it! In most text editors, you can save by navigating to File > Save or by pressing Command-S on MacOS or Ctrl-S on Windows.

Running tests

In SICP, we will use a program called `ok` to test our code. `ok` will be included in every assignment in this class.

Make sure you are in the `lab00` directory we created earlier (remember, the `cd` command lets you change directories).

In that directory, you can type `ls` to verify that there are the following three files:

- `lab00.py`: the starter file you just edited
- `ok`: our testing program
- `lab00.ok`: a configuration file for Ok

Now, let's test our code to make sure it works. You can run `ok` with this command:

```
python3 ok --local # remember to have the option --local
```

If you wrote your code correctly, you should see a successful test:

```
=====================================================================
Assignment: Lab 0
Ok, version v1.11.1
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Running tests

---------------------------------------------------------------------
Test summary
    1 test cases passed! No cases failed.
```

If you didn't pass the tests, `ok` will instead show you something like this:

```
---------------------------------------------------------------------
Doctests for twenty_twenty

>>> from lab00 import *
>>> twenty_twenty()
2013

# Error: expected
#     2020
# but got
#     2013

---------------------------------------------------------------------
Test summary
    0 test cases passed before encountering first failed test case
```

Fix your code in your text editor until the test passes.

> While `ok` is the primary assignment "autograder" in SICP, you may find it useful at times to write some of your own tests in the form of doctests. Then, you can try them out using the `-m doctest` option for Python.

Appendix: Useful Python command line options

When running a Python file, you can use options on the command line to inspect your code further. Here are a few that will come in handy. If you want to learn more about other Python command-line options, take a look at the documentation.

- Using no command-line options will run the code in the file you provide and return you to the command line.

  ```
  python3 [filepath]
  ```

- **`-i`**: The `-i` option runs your Python script, then opens an interactive session. In an interactive session, you run Python code line by line and get immediate feedback instead of running an entire file all at once. To exit, type `exit()` into the interpreter prompt. You can also use the keyboard shortcut `Ctrl-D` on Linux/Mac machines or `Ctrl-Z Enter` on Windows.

  If you edit the Python file while running it interactively, you will need to exit and restart the interpreter in order for those changes to take effect.

  ```
  python3 -i [filepath]
  ```

- **`-m doctest`**: Runs doctests in a particular file. Doctests are surrounded by triple quotes (`"""`) within functions.

  Each test in the file consists of `>>>` followed by some Python code and the expected output (though the `>>>` are not seen in the output of the doctest command).

  ```
  python3 -m doctest [filepath]
  ```