

Week 3

Evaluation, Variables and Turtle

Python Shell (Console)

The screenshot shows a Python 3.7.3 Shell window. The title bar says "Python 3.7.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell area displays the following Python session:

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> 5 + 3
8
>>> 'M' + 'iss' * 2 + 'ppi'
'Mississippi'
>>> x = 5 + 3
>>> x
8
>>>
```

Annotations with arrows point to specific lines:

- An orange box with a blue arrow points to the line "8" with the text "Echoes: “return values” of the previous line".
- A red box with a red arrow points to the line "x = 5 + 3" with the text "But sometimes there will NOT be any echo as there is NO “return value”".

At the bottom right of the shell window, it says "Ln: 10 Col: 4".

- However, this should NOT be the main area we work in (i.e. 90% of the time)

Arithmetic Evaluation

- What will be the evaluated values for the following:

$3 * 4 + 5$

$3 + 4 * 5$

$5 ** 3 \% 4$

$97 / 4$

$97 // 4$

You should try evaluating these WITHOUT typing into Python first

Logical Evaluation

- What will be the evaluated values for the following:

1 == 1

3 + 2 == 1 + 4

3 + 2 != 1 + 4

4 > 3

4 > 4

6 + 3 < 9 + 3

True or False

True and (False or True)

String Evaluation

- What will be the evaluated values for the following:

'abc' + 'def'

'gala' * 3

'mu' + 'ha' * 4

('ba' *2+'bidu'*2+'bi' + 'jam' *2)*3

'banana'[3]

'banana'[2:4]

'banana'[1::2]

String Slicing

- Let $s = \text{'abcdef'}$
- What is the result of $s[]$ and $s[:2]$ and $s[:2:]$?
 - Are they the same?
- Only $s[:2]$ and $s[:2:]$ are the same.
- $s[]$ is a syntax error

| | | | | | | |
|-------|---|---|---|---|---|---|
| | a | b | c | d | e | f |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

Start – By default, start from index 0.
Stop – By default, include the last letter.
Step – By default, “jump” by 1 step.

String Slicing

- Let $s = \text{'abcdef'}$
- What about $s[5:0:-1]$?
‘fedcb’
- What happens if we do $s[:2:-1]$?
‘fed’
- Lecture example: $s[::-1]$
‘fedcba’

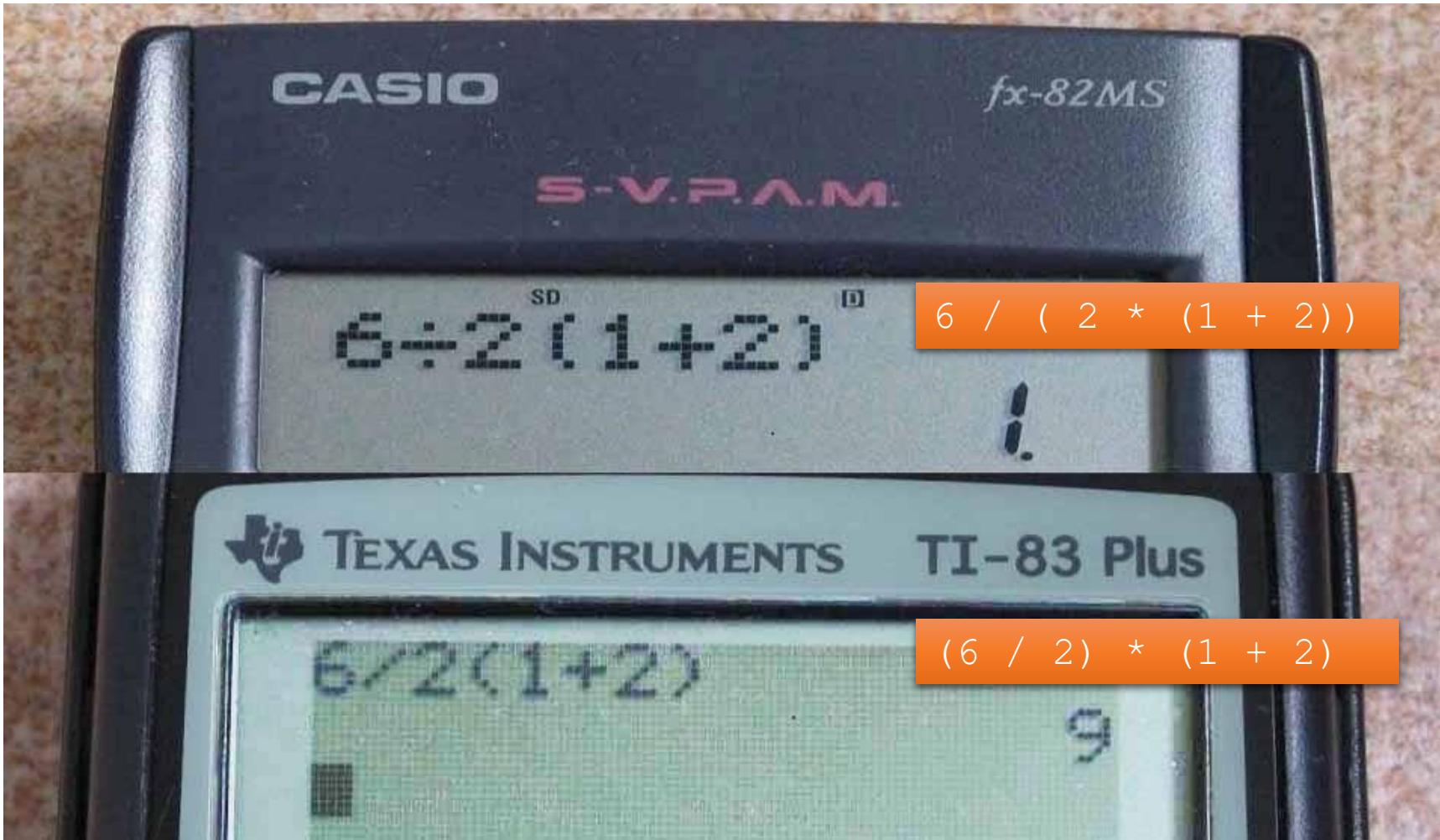
| | | | | | | |
|-------|---|---|---|---|---|---|
| | a | b | c | d | e | f |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

Start – By default, start from index 0.
Stop – By default, include the last letter.
Step – By default, “jump” by 1 step.

Default

- If step > 0
 - Start – By default, start from index 0.
 - Stop – By default, include the last letter.
 - Step – By default, “jump” by 1 step.
- Else (step < 0)
 - Default start = last letter
 - Default end = -n-1
- Let n = length of your string
- If step > 0
 - Start = 0
 - Stop = n
- Else if step < 0
 - Start = n-1
 - Stop = -n-1

Operator Precedence



Python Operator Precedence

- $6/2 * (1+2)$
- $6/2 * (1+2)$
- $6/2 * 3$
- $3*3$
- 9

| Operators | Associativity |
|---|---------------|
| () Highest precedence | Left - Right |
| ** | Right - Left |
| +x , -x, ~x | Left - Right |
| * , / , // , % | Left - Right |
| + , - | Left - Right |
| << , >> | Left - Right |
| & | Left - Right |
| ^ | Left - Right |
| | Left - Right |
| Is, is not, in, not in, <, <= , > , >= , == , != | Left - Right |
| Not x | Left - Right |
| And | Left - Right |
| Or | Left - Right |
| If else | Left - Right |
| Lambda | Left - Right |
| =, +=, -=, *=, /= Lowest Precedence | Right - Left |

How Do I Remember It All ... ? BODMAS !

- B** Brackets first
- O** Orders (i.e. Powers and Square Roots, etc.)
- DM** Division and Multiplication (left-to-right)
- AS** Addition and Subtraction (left-to-right)

Divide and Multiply rank equally (and go left to right).

Add and Subtract rank equally (and go left to right)

Arithmetic Evaluation

- What will be the evaluated values for the following
(or what is the orders of the operators?)

1 + 2 * 3

1 + 2 * 3 ** 4

1 + 2 * 3 ** 4 - 5

not 0 + 1

What is the difference?

- What do we have when we ask if 1 is it the same as '1'?

1 == '1'

- Or what is the difference between the following two lines?

123+456

'123' + '456'

Type Conversions

```
>>> type(123)
<class 'int'>
>>> 123 + 456
579
>>> type('123')
<class 'str'>
>>> '123' + '456'
'123456'
>>> '123' + 456
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    '123' + 456
TypeError: can only concatenate str (not "int") to str
```

A diagram consisting of two blue arrows originating from the '+' operator in the second and fourth lines of code, pointing towards a callout box. The callout box has a light orange background and contains the text: "Note that the "+" operator performs differently for different types".

Variables

- Now you should know the following:

3 * 4 + 5

3 + 4 * 5

- How about

x = 3

y = 4

z = 5

x * y + z

“Creation” of Variables

- What will be the evaluated values for the following:

a * b + c

- Error! Why?
- Because a, b and c are undeclared
 - In other words, “not created”, “not born yet”
 - Whenever you type a line

a = (something)
 - A variable (a) is born

From scratch

m + 3

- Error!

m = 1

m + 3



Creation of the variable m

- Output:

4

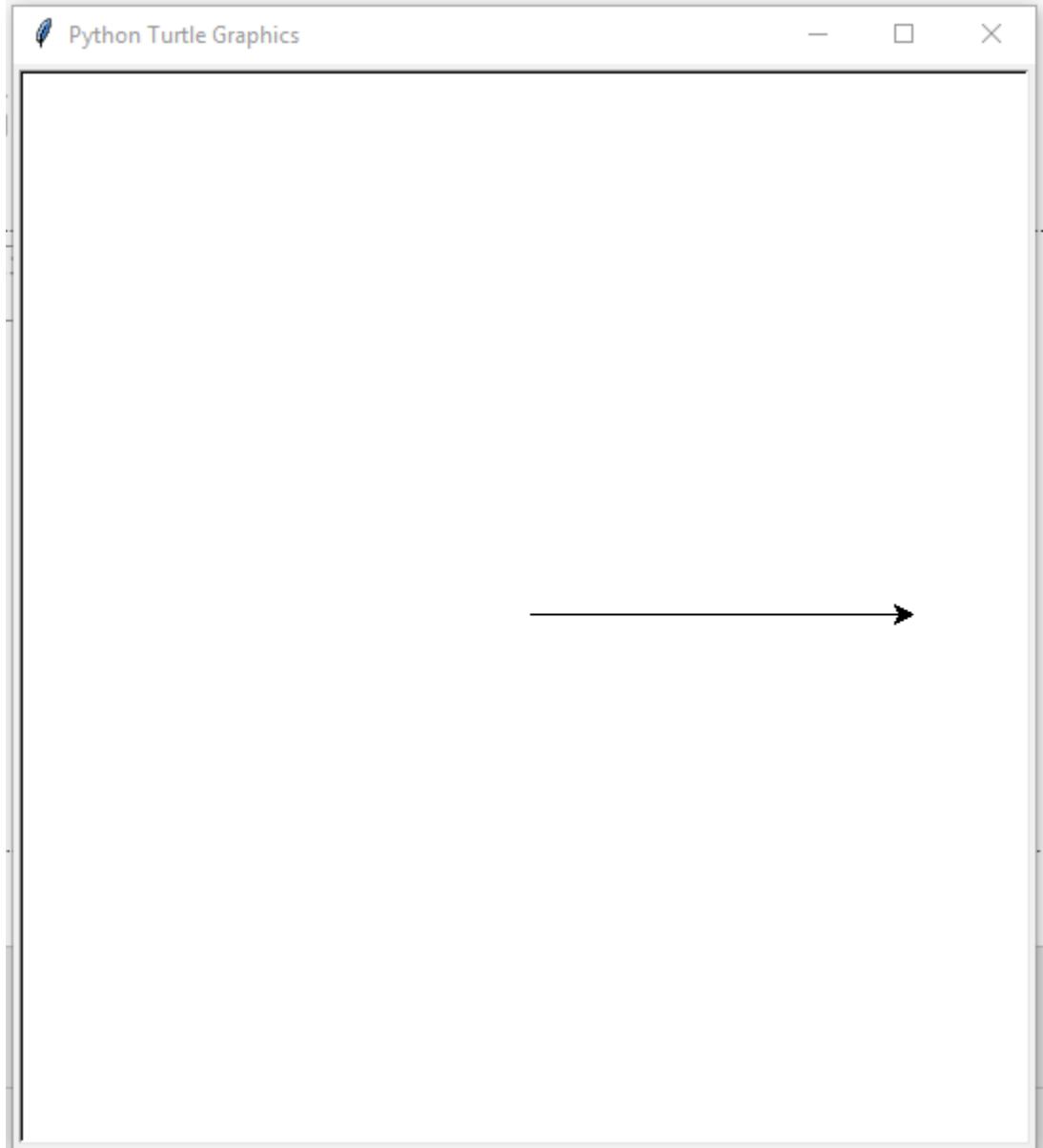
Turtle Graphics

```
>>> from turtle import *
>>> forward(100)
>>>
```

- Or you can use the short from

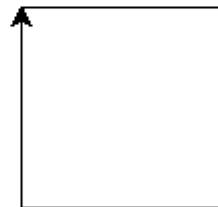
```
>>> from turtle import *
>>> fd(100)
```

- Or this,
- ```
>>> import turtle
>>> turtle.fd(100)
```
- But for our course, please do  
**NOT** use the last form



# Turtle Graphics

```
>>> from turtle import *
>>> fd(100)
>>> rt(90)
>>> fd(100)
>>> rt(90)
>>> fd(100)
>>> rt(90)
>>> fd(100)
>>> rt(90)
>>> fd(100)
>>>
```



# More Turtle Commands

- You can go to the website:  
<https://docs.python.org/3.3/library/turtle.html?highlight=turtle>
- Or just google “Python Turtle”

Python » 3.3.7 Documentation » The Python Standard Library » 24. Program Frameworks »

---

Table Of Contents

- 24.1. `turtle` — Turtle graphics
  - 24.1.1. Introduction
  - 24.1.2. Overview of available Turtle and Screen methods
    - 24.1.2.1. Turtle methods
    - 24.1.2.2. Methods of `TurtleScreen`/`Screen`
  - 24.1.3. Methods of `RawTurtle`/`Turtle` and corresponding functions
    - 24.1.3.1. Turtle motion
    - 24.1.3.2. Tell Turtle's state
    - 24.1.3.3. Settings for measurement
    - 24.1.3.4. Pen control
      - 24.1.3.4.1.

## 24.1. `turtle` — Turtle graphics

### 24.1.1. Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feuerstein and Seymour Papert at the Massachusetts Institute of Technology. It has been ported to Python by John Hunter and others.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an `import turtle`, give it the command `turtle.forward(15)`, and it moves (on-the-fly), drawing a line as it moves. Give it the command `turtle.right(25)`, and it rotates in-place 25 degrees clockwise.

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The `turtle` module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the `-n` switch.

The `turtle` module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways.

**Turtle star**

Turtle can draw intricate shapes using simple moves.



- Actually, this is what most programmers do

# Functions

# Let's Write Our Own Function!

Define (keyword)      Function name      Input (Argument)

```
def square(x):
 return x * x
```

Indentation      Output

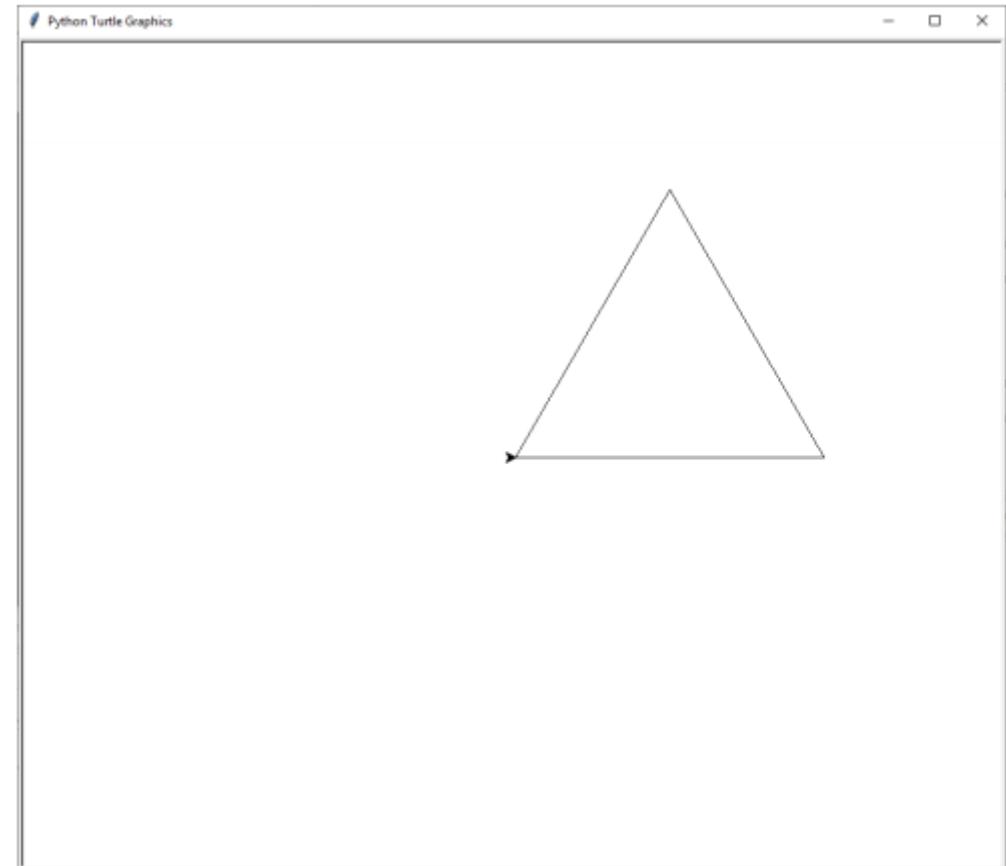
The diagram illustrates the structure of a Python function definition. It features five main labels: 'Define (keyword)', 'Function name', 'Input (Argument)', 'Indentation', and 'Output'. Blue arrows point from these labels to specific parts of the code. 'Define (keyword)' points to the 'def' keyword. 'Function name' points to the word 'square'. 'Input (Argument)' points to the variable 'x' in the argument list. 'Indentation' points to the two-line indentation at the beginning of the function body. 'Output' points to the 'return x \* x' statement.

# Put Statements into a Function

- For the Assignment last week
- Your answer will be something like (yours maybe a bit different)

```
fd(300)
lt(120)
fd(300)
lt(120)
fd(300)
lt(120)
```

- But if I want to draw it again?
  - It's too troublesome to type the above lines again and again



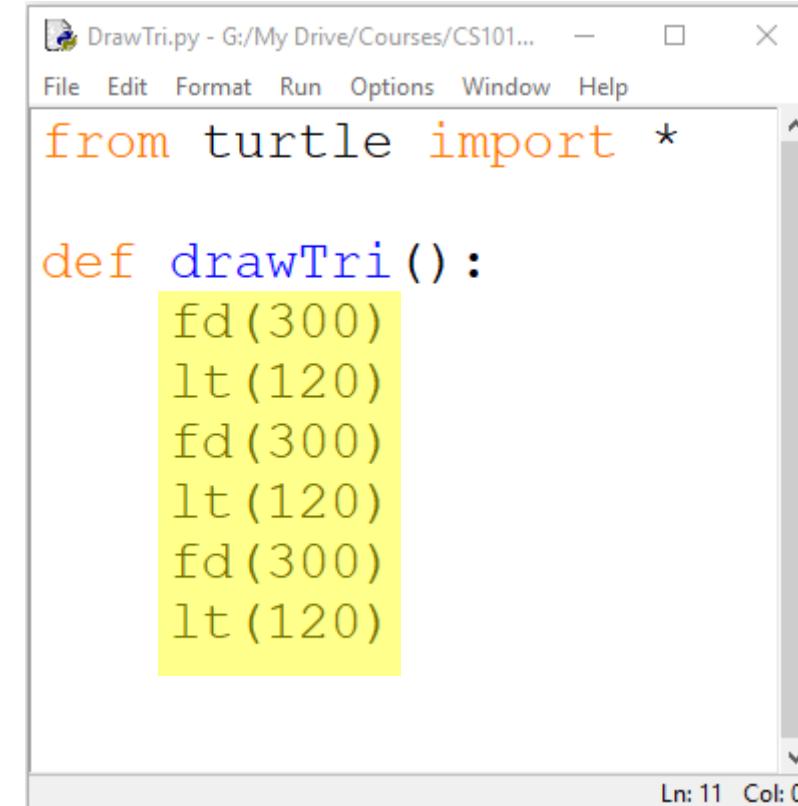
# Put Statements into a Function

- Last week Assignment
- Your answer will be something like (yours maybe a bit different)

```
fd(300)
lt(120)
fd(300)
lt(120)
fd(300)
lt(120)
```

- But if I want to draw it again?
  - It's too troublesome to type the above lines again

- We save it into a file by
  - In IDLE, File > New



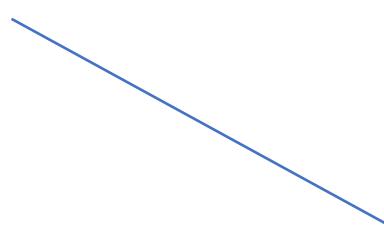
The screenshot shows the IDLE Python editor window with a script named "DrawTri.py". The code in the editor is:

```
from turtle import *

def drawTri():
 fd(300)
 lt(120)
 fd(300)
 lt(120)
 fd(300)
 lt(120)
```

The status bar at the bottom right indicates "Ln: 11 Col: 0".

# Put Statements into a Function

- After you saved the file and run it
  - You can call the function `drawTri()` by  
`>>> drawTri()`
  - Or
    - Directly put it into the file
- 

- We save it into a file by
  - In IDLE, File > New

```
from turtle import *

def drawTri():
 fd(300)
 lt(120)
 fd(300)
 lt(120)
 fd(300)
 lt(120)

drawTri()
```

# Function Parameters

- What if we want to draw a triangle that is larger or smaller
  - Namely, the side length is different from 300?
  - Do we write...

```
def drawTri():
 fd(200)
 lt(120)
 fd(200)
 lt(120)
 fd(200)
 lt(120)
```

```
def drawTri():
 fd(100)
 lt(120)
 fd(100)
 lt(120)
 fd(100)
 lt(120)
```

```
from turtle import *

def drawTri():
 fd(300)
 lt(120)
 fd(300)
 lt(120)
 fd(300)
 lt(120)

drawTri()
```

- Etc...?

# Capture the COMMON Pattern

- What if we want to draw a triangle that is larger or smaller
  - Namely, the side length is different from 300?
  - Do we write...

```
def drawTri():
 fd(200)
 lt(120)
 fd(200)
 lt(120)
 fd(200)
 lt(120)
```

```
def drawTri():
 fd(100)
 lt(120)
 fd(100)
 lt(120)
 fd(100)
 lt(120)
```

- Etc...?

- No, we **capture the common pattern** and make it an input of the function

```
from turtle import *

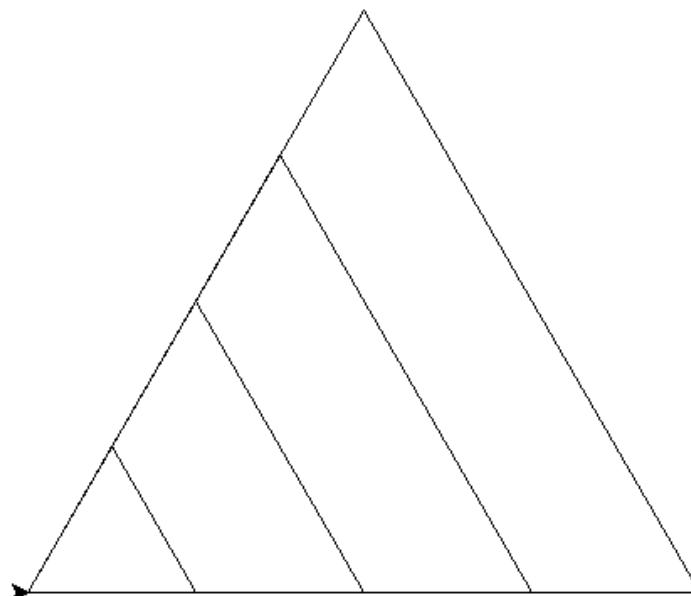
def drawTri(length):
 fd(length)
 lt(120)
 fd(length)
 lt(120)
 fd(length)
 lt(120)

drawTri(100)
drawTri(200)
drawTri(300)
```

This is an important skill in computational thinking

# Capture the COMMON Pattern

```
>>> drawTri(100)
>>> drawTri(200)
>>> drawTri(300)
>>> drawTri(400)
>>>
```



This is an important skill in computational thinking

- No, we capture the common pattern and make it an input of the function

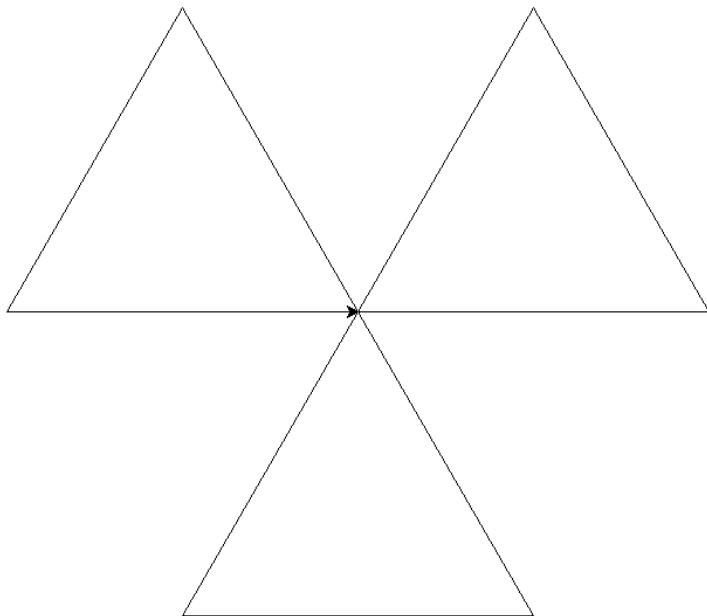
```
from turtle import *

def drawTri(length):
 fd(length)
 lt(120)
 fd(length)
 lt(120)
 fd(length)
 lt(120)

drawTri(100)
drawTri(200)
drawTri(300)
```

# Moreover

- What does this code do?
  - Output:



```
from turtle import *

def drawTri(length):
 fd(length)
 lt(120)
 fd(length)
 lt(120)
 fd(length)
 lt(120)

def foo():
 drawTri(100)
 lt(120)
 drawTri(100)
 lt(120)
 drawTri(100)
 lt(120)

foo()
```