Python Language Rules

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Python

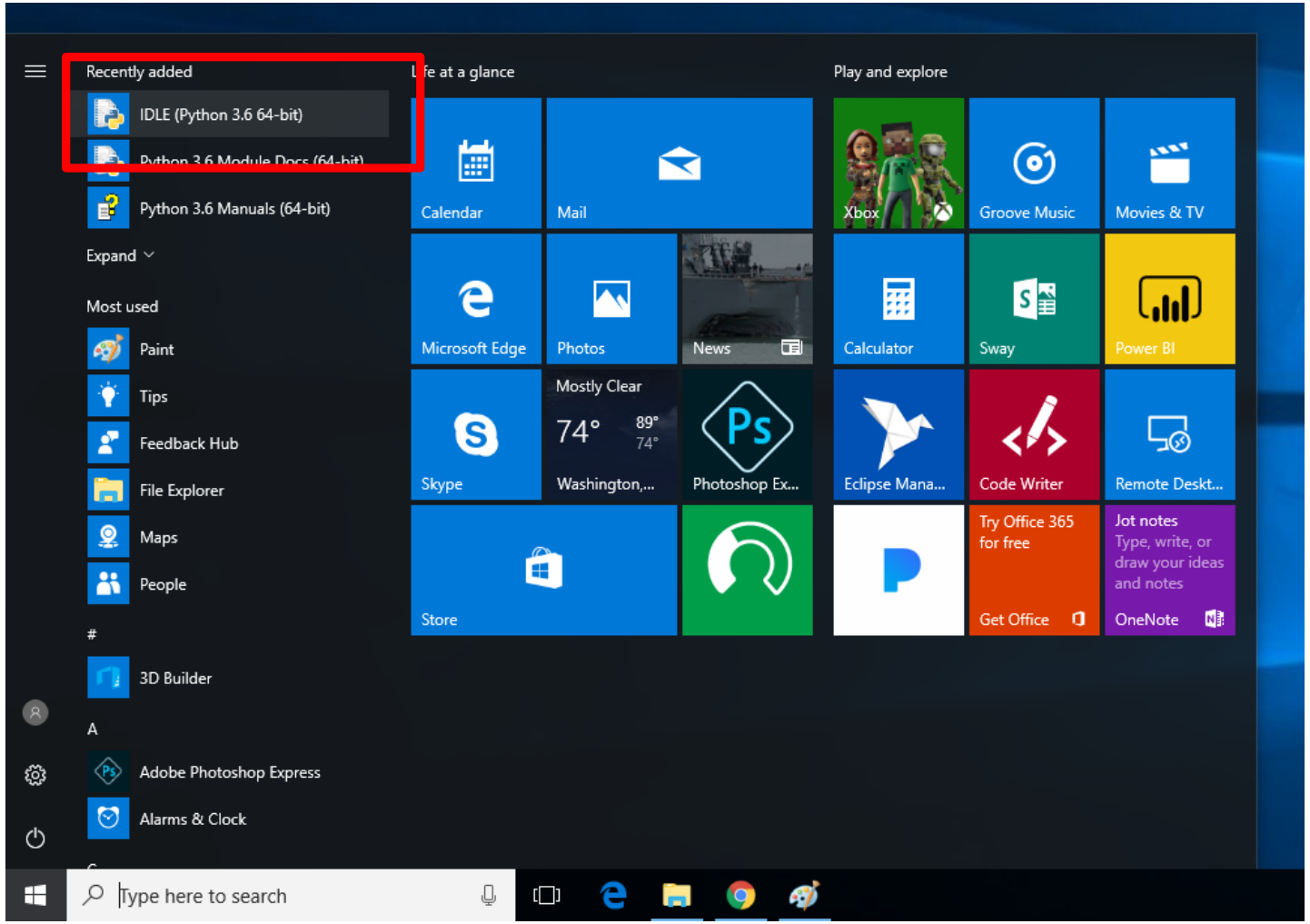
Philosophy of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

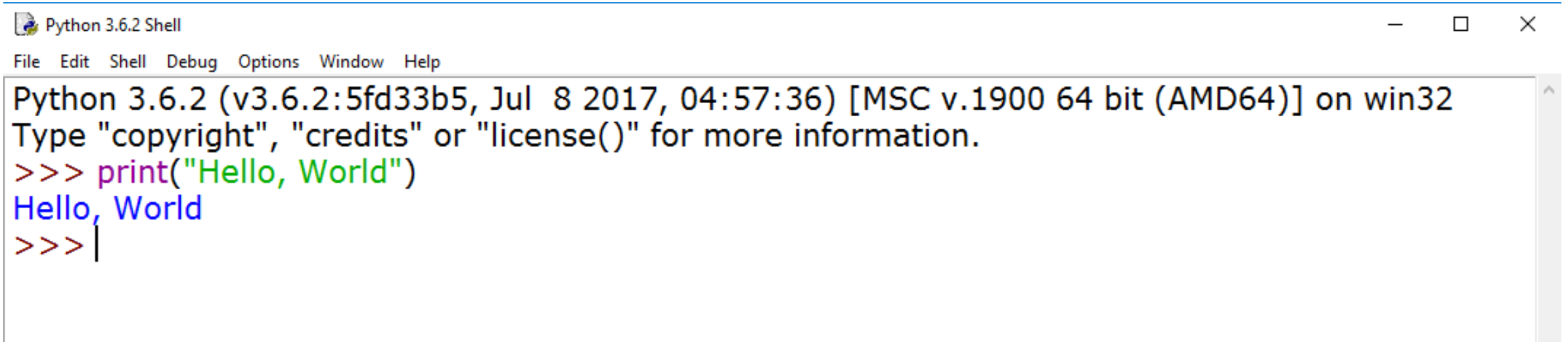
Starting the Python Interpreter

- IDLE (Python GUI) Program



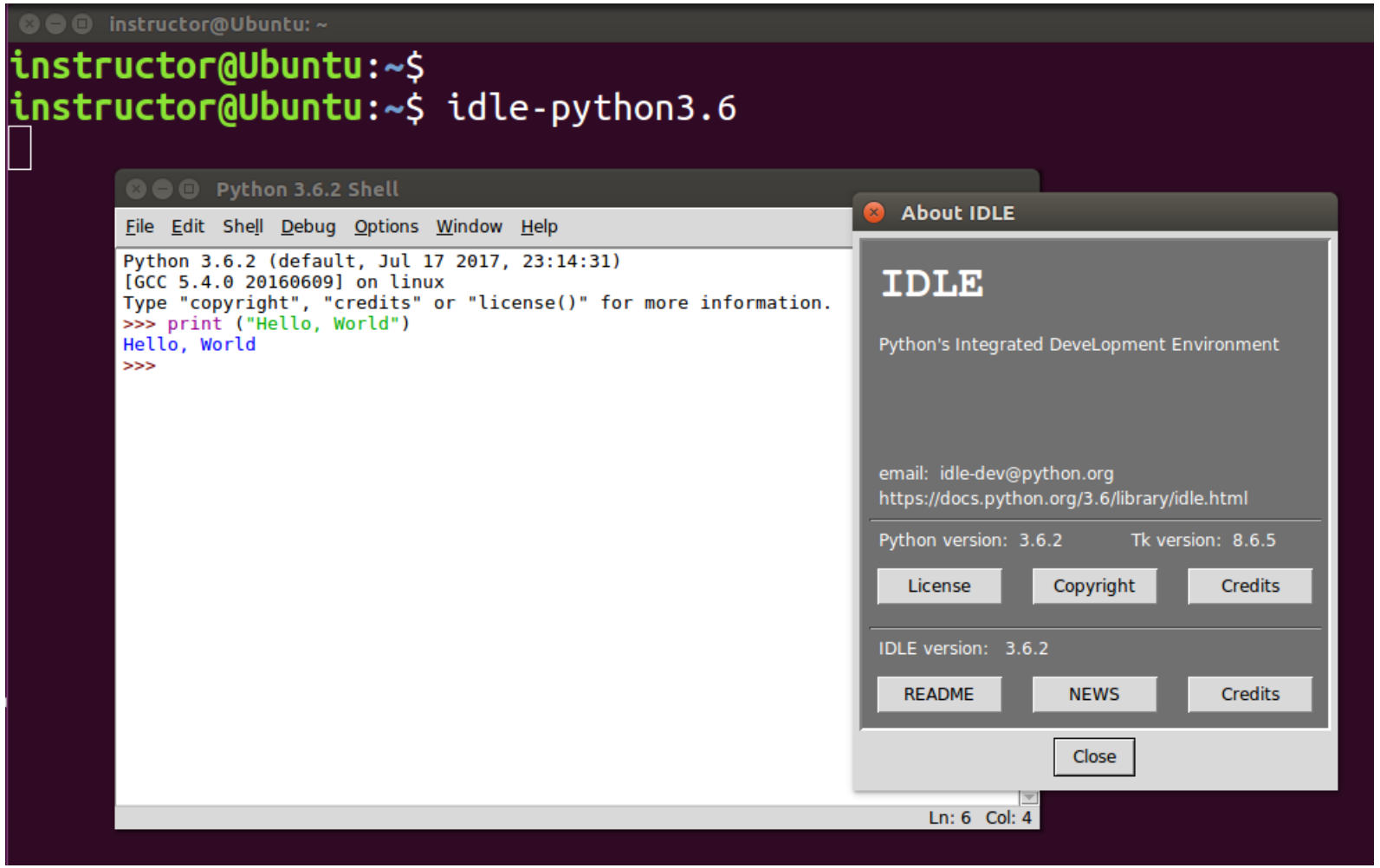
Starting the Python Interpreter (Cont.)

- IDLE (Python GUI) Program



Starting the Python Interpreter (Cont.)

- IDLE (Python GUI) Program

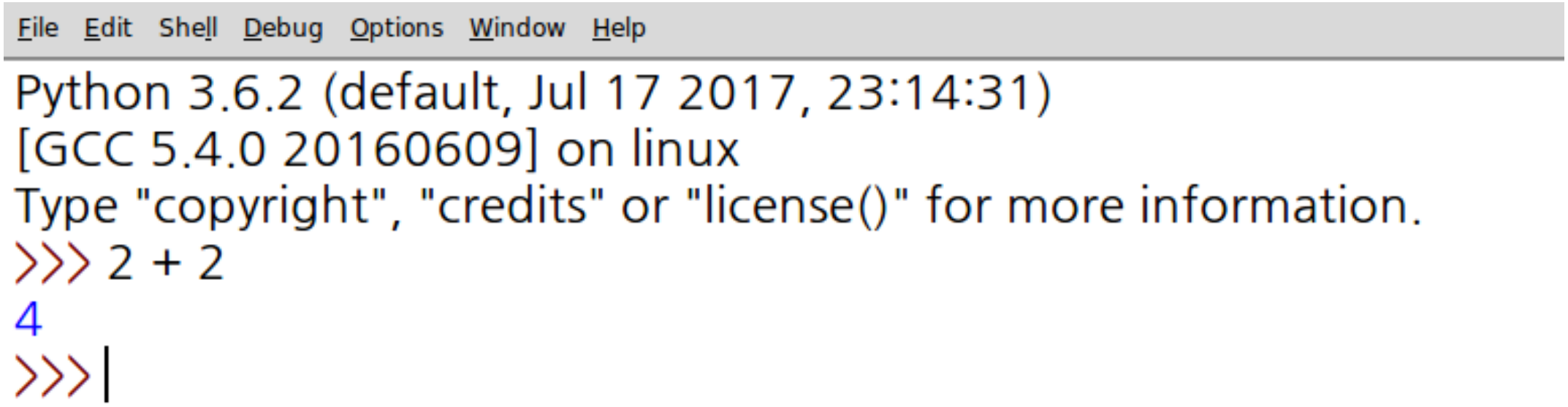


Starting the Python Interpreter (Cont.)

- IDLE
  - **I**nteractive **D**eve**L**opment **E**nvironment
  - Program that helps us type in our own programs and games.
  - Interactive shell
    - First run IDLE window.
    - Can work just like a calculator.

Starting the Python Interpreter (Cont.)

- Some simple math stuff
  - Type `2 + 2` into the shell and press the **Enter key**.
  - Computer should respond with the number `4`.
    - : the sum of `2 + 2`

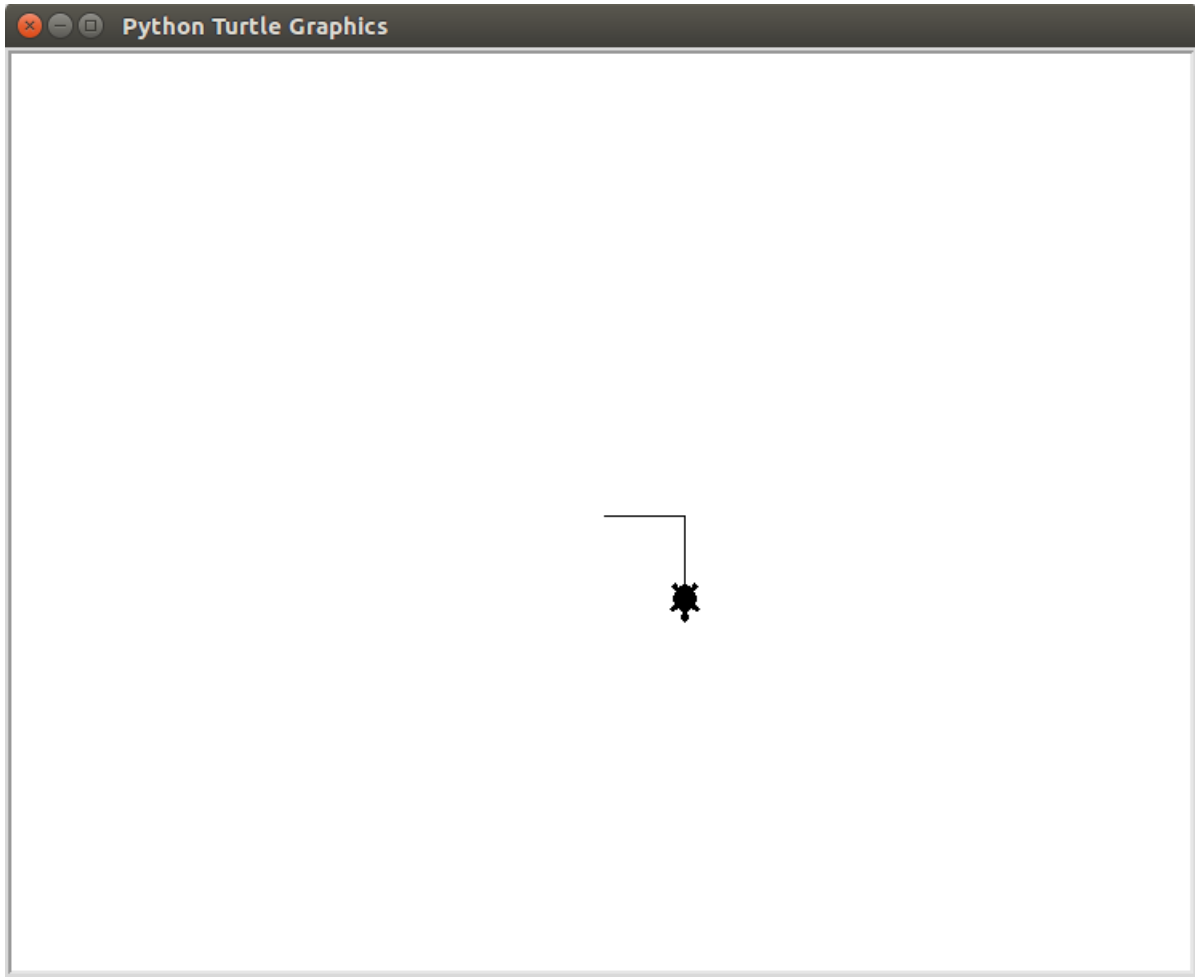


Python IDLE - Lab

Python 3.6.2 Shell

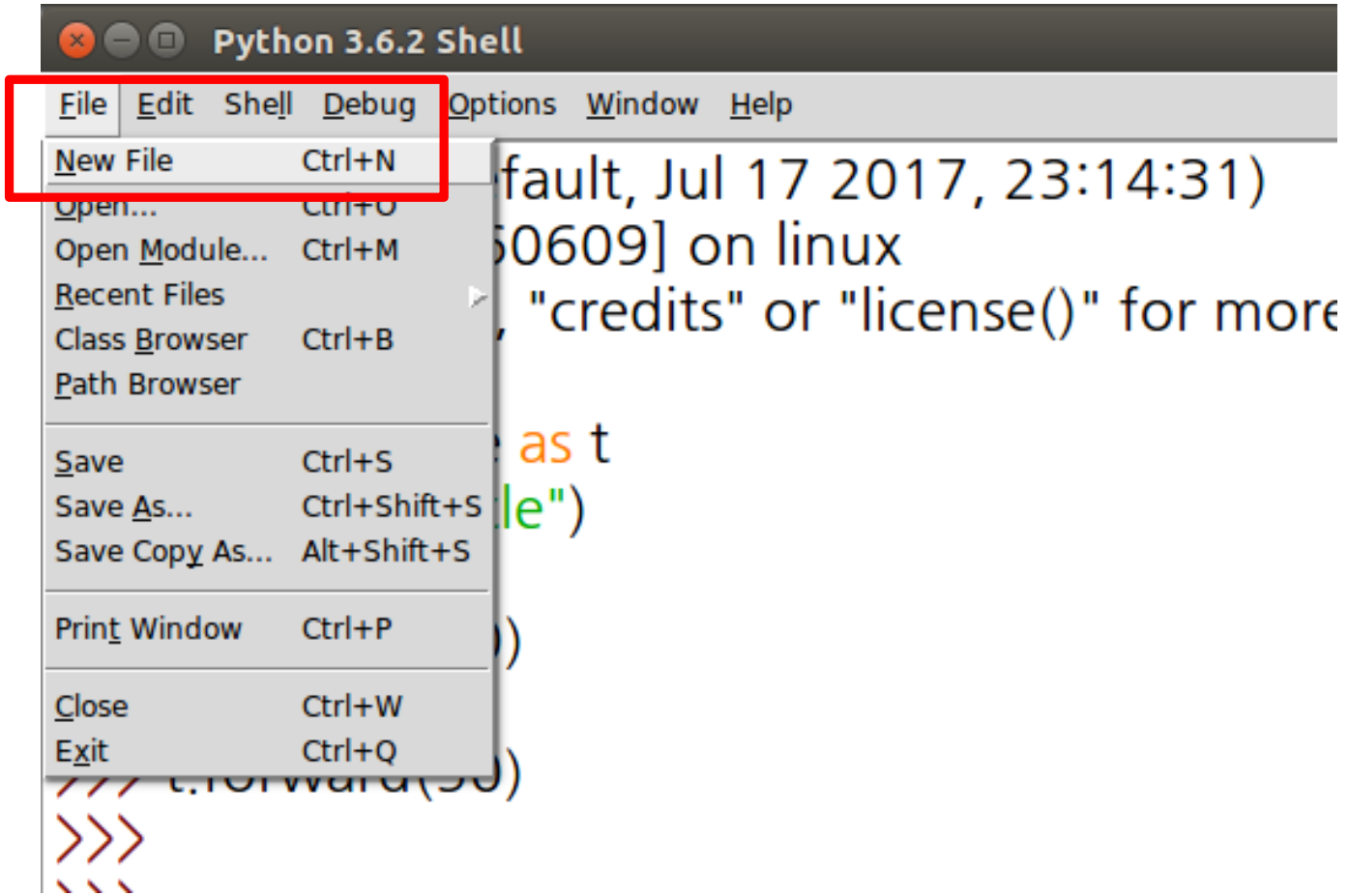
File Edit Shell Debug Options Window Help

```
Python 3.6.2 (default, Jul 17 2017, 23:14:31)
[GCC 5.4.0 20160609] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
>>> import turtle as t
>>> t.shape("turtle")
>>>
>>> t.forward(50)
>>> t.right(90)
>>> t.forward(50)
>>>
```



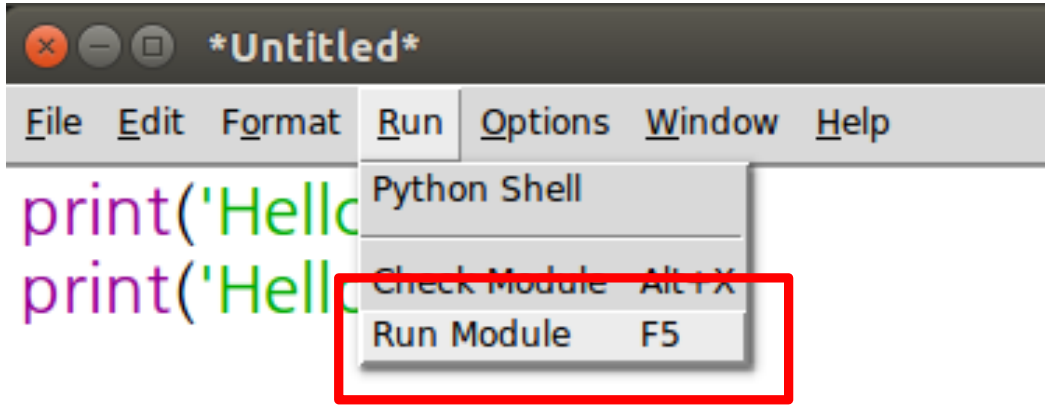
Python IDLE – Lab (Execute with File)

1. File > New File



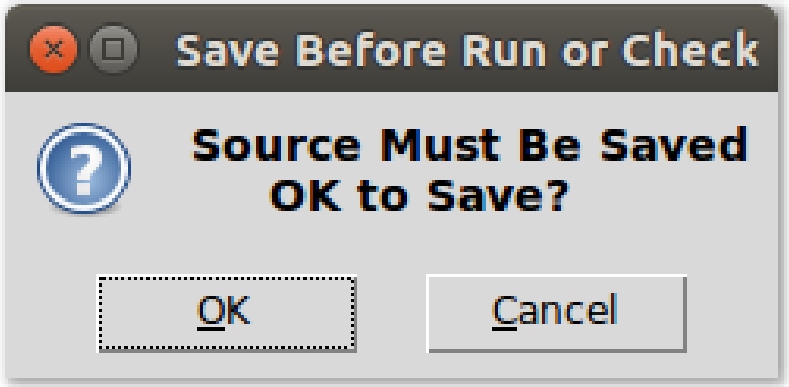
Python IDLE – Lab (Execute with File) (Cont.)

2. Source Coding
3. **Run** > **Run Module**

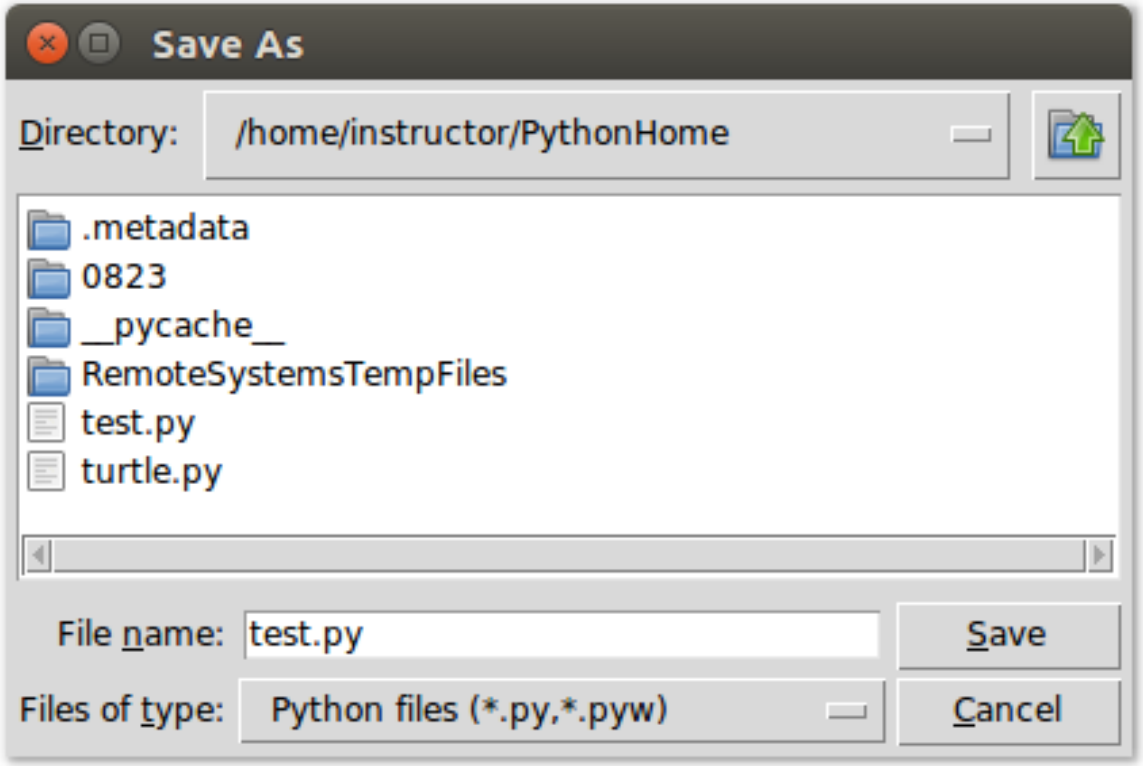


Python IDLE – Lab (Execute with File) (Cont.)

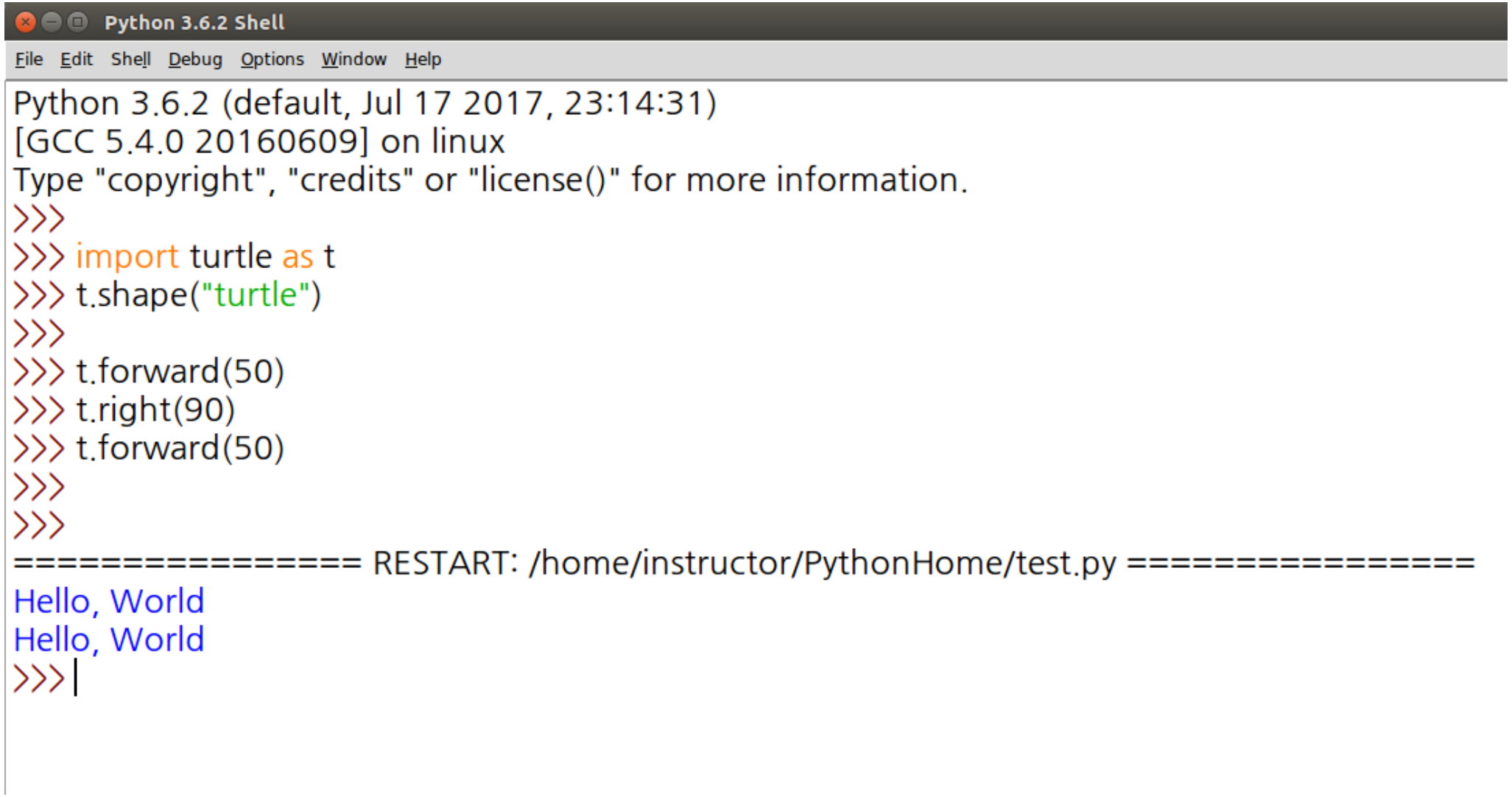
4. Click OK to save file



5. Save the file



Python IDLE – Lab (Execute with File) (Cont.)



Python IDLE – Lab

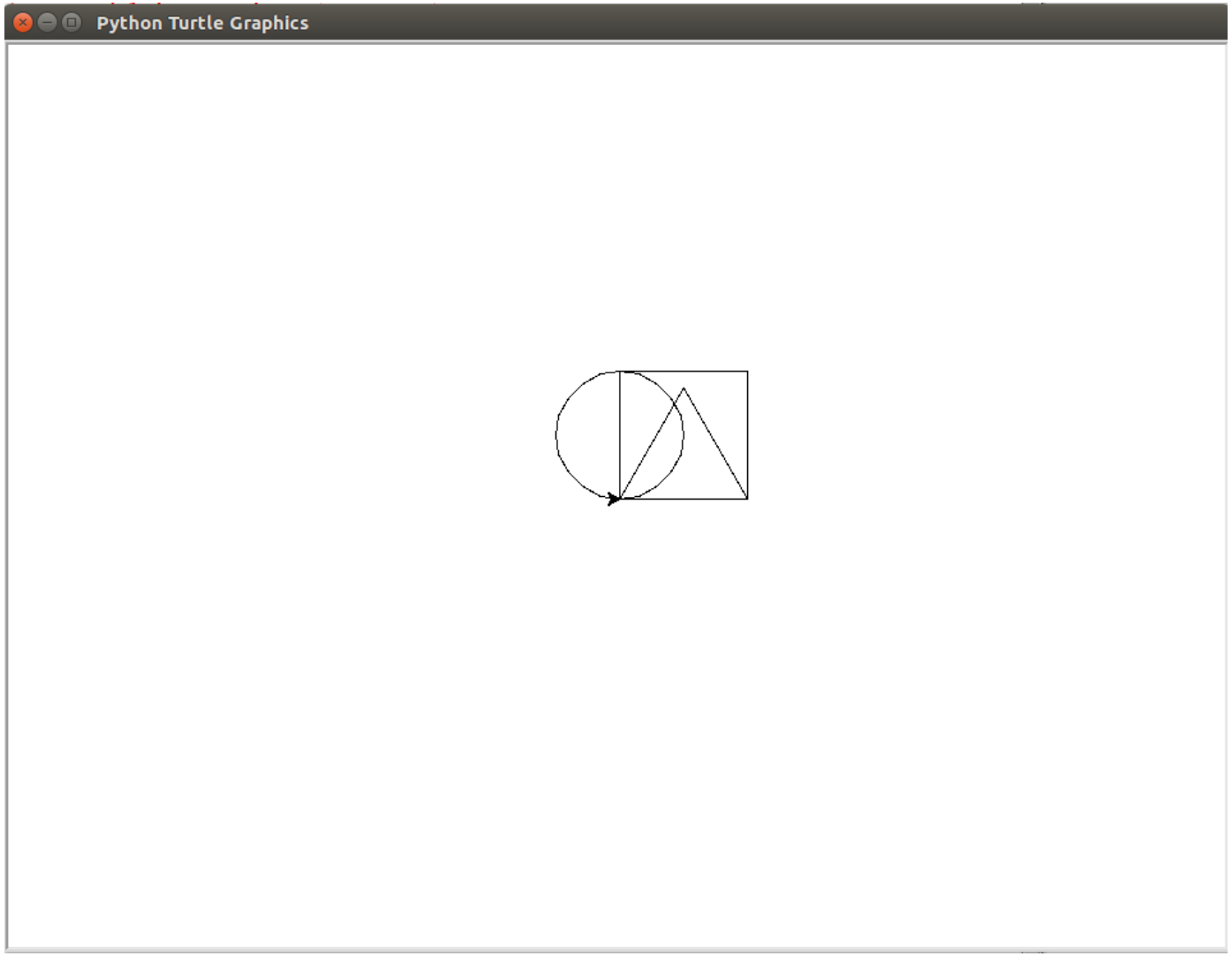
```
#myturtle.py

import turtle as t

#삼각형 그리기
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)

#사각형 그리기
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)

#원 그리기기
t.circle(50)
```



Python IDLE – Lab (Cont.)

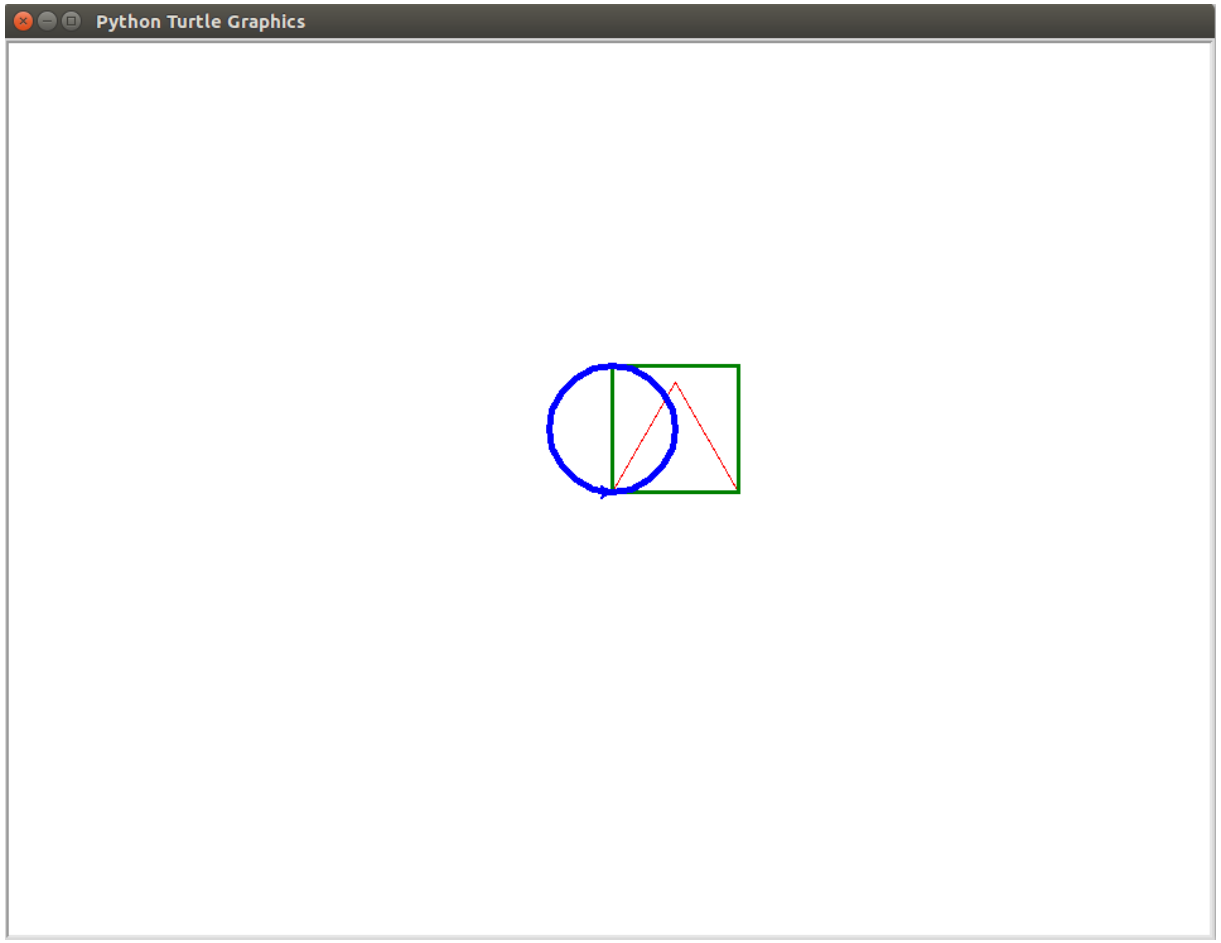
```
#myturtle_color.py

import turtle as t

#삼각형 그리기
t.color("red")
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
```

```
#사각형 그리기
t.color("green")
t.pensize(3)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)

#원 그리기기
t.color("blue")
t.pensize(5)
t.circle(50)
```



Python Syntax Features

- Indentation

```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"]'
    else:
        print '"];'
        children = []
        for  in n, childenumerate(ast[1:]):
            children.append(dotwrite(child))
        print ,'    %s -> {' % nodename
        for  in :namechildren
            print '%s' % name,
```

Python Syntax Features (Cont.)

- Indentation

Python

```
def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(x - 1)
```

C

```
int factorial(int x)
{
    if(x == 0)
    {
        return 1
    }
    else
    {
        return x     actorial(x - 1);
    }
}
```

```
int factorial(int x) {
 if(x == 0) {return 1;} else
 {return x * factorial(x - 1); } }
```

Python Syntax Features (Cont.)

- String Indexing and Slicing

```
>>> t = "This is Python Class, And 1st day"
>>> t[2]
'i'
```

Python Syntax Features (Cont.)

- String Indexing and Slicing

```
>>> t[2]
'i'
>>> t[6]
's'
>>> t[1 2]
'o'
>>> t[2 1]
' '
>>> t[-1]
'y'
```

Python Syntax Features (Cont.)

- String Indexing and Slicing

```
>>> t[1:3]
'hi'
>>> t[2:10]
'is is Py'
```

Python Syntax Features (Cont.)

- String Indexing and Slicing

```
>>> t = "2015-03-10 14:21:35"
>>> date = t[:10]
>>> time = t[11:]
>>> print(date)
2015-03-10
>>> print(time)
14:21:35
```

Python Syntax Features (Cont.)

- String Formatting

```
>>> name = "Python"
>>> print("This is %s" % name)
This is Python
>>> print("This is %d" % 10)
This is 10
```

Python Syntax Features (Cont.)

- 한글처리 in Python

```
text = "한글 출력 테스트"
print(text)
```

Python Identifiers

- Is a name used to identify a variable, function, class, module or other object.
- Start with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @, $, and % within identifiers.
- Python is a ***case sensitive*** programming language.

Python Identifiers (Cont.)

- Naming conventions for Python identifiers :
  - Class names start with **an uppercase letter**.
  - All other identifiers start with a lowercase letter.
  - Starting an identifier with a single leading underscore(_) indicates that the identifier is ***private***.
  - Starting an identifier with two leading underscores(__) indicates a ***strong private*** identifier.
  - If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

- Cannot use as constants or variables or any other identifier names.
- All the Python keywords contain lowercase letters only.

```
>>>
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finall
y', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', '
while', 'with', 'yield']
>>>
```

Reserved Words (Cont.)

and	as	assert	break	class
continue	def	del	elif	else
except	exec	finally	for	from
global	if	import	in	is
lambda	not	or	pass	print
raise	return	try	while	with
yield	False	True	None	nonlocal

Lines and Indentation

- Python does not use braces({ }) to indicate blocks of code for class and function definitions or flow control.
- Blocks of code are denoted by line indentation, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

```
if True :
    print ("True")
else :
    print ("False")
```

Multi-Line Statements

- Statements in Python typically end with a new line.
- However, allows the use of the line continuation character (\) to denote that the line should continue.

```
>>> total = item_one + \
            item_two + \
            item_three
```

Quotation in Python

- Python accepts single (') , double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines.

```
>>> word = 'word'
>>> sentence = "This is a sentence."
>>> paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
>>>
```

Comments in Python

- A hash sign (#) that is not inside a string literal is the beginning of a comment.
- All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

```
>>> # First comment
>>> print ("Hello, Python!")  # second comment
Hello, Python!
>>>
```

Comments in Python (Cont.)

- You can type a comment on the same line after a statement or expression.

```
>>> name = "Orange"  # This is again comment
>>>
```

- Python does not have multiple-line commenting feature.

```
>>> # This is a comment.
>>> # This is a comment, too.
>>> # This is a comment, too.
>>> # I said that already.
>>>
```

Using Blank Lines

- A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.
- In an interactive interpreter session, must enter an empty physical line to terminate a multiline statement.

Waiting for the User

- The following line of the program displays the prompt and, the statement saying **"Press the enter key to exit"**, and then waits for the user to take action.

```
>>>
>>> input ("₩n₩nPress the enter key to exit.")


Press the enter key to exit.
''
>>>
```

Multiple Statements on a Single Line

- The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block.
- Here is a sample snip using the semicolon.

```
>>>
>>> import sys;  x = 'foo' ;  sys.stdout.write(x + '₩n')
foo
4
>>>
```

Multiple Statement Groups as Suites

- Groups of individual statements, which make a single code block are called suites in Python.
- Compound or complex statements, such as if, while, def, and class require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

Variables, Names, and Objects

- In Python, everything—booleans, integers, floats, strings, even large data structures, functions, and programs—is implemented as an object.
- An object is like a clear plastic box that contains a piece of data.

Variables, Names, and Objects (Cont.)

- Variables - The values put into variables can be changed at any time.
- Constants - The values put into constants cannot be changed.

Variables, Names, and Objects (Cont.)

- Assigning Values to Variables
  - Python variables do not need explicit declaration to reserve memory space.
  - The declaration happens automatically when you assign a value to a variable.
  - The equal sign (=) is used to assign values to variables.
  - The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Variables, Names, and Objects (Cont.)

```
>>> counter = 100  # An integer assignment
>>> miles = 1000.0 # A floating point
>>> name = "John" # a string
>>>
>>> print(counter)
100
>>> print(miles)
1000.0
>>> print(name)
John
>>>
```

Variables, Names, and Objects (Cont.)

- Multiple Assignment
  - Python allows you to assign a single value to several variables simultaneously.

```
>>> a = b = c = 1
```

Variables, Names, and Objects (Cont.)

- Multiple Assignment
  - Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location.
  - You can also assign multiple objects to multiple variables.

```
>>> a, b, c = 1, 2, "John"
>>>
>>> print(a)
1
>>> print(b)
2
>>> print(c)
John
>>>
```

Standard Data Types

- The data stored in memory can be of many types.
- For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.
- Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Standard Data Types (Cont.)

- Python has five standard data types :
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary

Python Numbers

- Number data types store numeric values.
- Number objects are created when you assign a value to them.

```
>>> var1 = 1
>>> var2 = 10
```

- You can also delete the reference to a number object by using the `del` statement.

```
>>> del var1[, var2[, var3[…., varN]]]
```

Python Numbers (Cont.)

- Can delete a single object or multiple objects by using the **`del`** statement.

```
>>> del var
>>> del var_a, var_b
```

Python Numbers (Cont.)

- Python supports three different numerical types :
  - `int` (signed integers)
  - `float` (floating point real values)
  - `complex` (complex numbers)
- All integers in Python3 are represented as **long** integers.
- Hence, there is no separate number type as long.

Python Numbers (Cont.)

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

A complex number consists of an ordered pair of real floating-point numbers denoted by `x + yj`, where `x` and `y` are real numbers and `j` is the imaginary unit.

Python Numbers (Cont.) – Lab

```
1  print(2 + 3)
2
3  print(2 * 3)
4
5  print(3 / 2)
6
7  print(3 ** 2)
8
9  print(2 + 3 * 4)
10
11 print(2 * 0.1)
12
13 print(0.1 + 0.1)
14
```

Python Numbers (Cont.) – Lab

```
>>> a = 3
>>> b = 123456789
>>> c = 12345678901234567890123456789012345678890
>>> print (a)
3
>>> print (b)
123456789
>>> print (c)
12345678901234567890123456789012345678890
>>> type(c)
<class 'int'>
>>>
>>> f = 22 / 7
>>> f
3.142857142857143
>>> type(f)
<class 'float'>
>>>
```

Python Numbers (Cont.) – Lab

```
>>> hex(0)
'0x0'
>>>
>>> hex(255)
'0xff'
>>>
>>> a = 0xFF
>>> a
255
>>>
>>> b= 0x20
>>> b
32
>>> c = 0x0
>>> c
0
>>>
```

```
>>> bin(0)
'0b0'
>>>
>>> bin(8)
'0b1000'
>>>
>>> bin(32)
'0b100000'
>>>
>>> bin(255)
'0b11111111'
>>>
>>> a = 0b100
>>> a
4
>>> b = 0b1001
>>> b
9
>>> c = 0b11111111
>>> c
255
>>>
```

Python Numbers (Cont.) – Lab

```
>>> oct(8)
'0o10'
>>> oct(10)
'0o12'
>>> oct(64)
'0o100'
>>>
>>> a = 0o10
>>> a
8
>>> b = 0o12
>>> b
10
>>> c = 0o100
>>> c
64
>>>
```

```
>>> a = 1.23 + 0.32
>>> a
1.55
>>> b = 3.0 - 1.5
>>> b
1.5
>>> c = 2.1 * 2.0
>>> c
4.2
>>> d = 4.5 // 2.0
>>> d
2.0
>>> e = 4.5 % 2.0
>>> e
0.5
>>> f = 4.5 / 2.0
>>> f
2.25
>>>
>>> g = 43.2 - 43.1
>>> g
0.10000000000000142
>>> #부동 소수형의 정밀도의 한계
```

Python Numbers (Cont.) – Lab

```
>>> a = 2 + 3j
>>> a
(2+3j)
>>> type(a)
<class 'complex'>
>>>
>>> a.real
2.0
>>> a.imag
3.0
>>> a.conjugate()
(2-3j)
>>>
```

```
>>> a = (1 + 2j) + (3 + 4j)    # (a + bj) + (c + dj) = (a + c) + (b + d)j
>>> a
(4+6j)
>>>
>>> b = (1 + 2j) - (3 + 4j)
>>> b
(-2-2j)
>>> c = (1 + 2j) * (3 + 4j)
>>> c
(-5+10j)
>>> d = (1 + 2j) / (3 + 4j)
>>> d
(0.44+0.08j)
>>>
```

Python Strings

- Strings in Python are identified as a contiguous set of characters represented in the quotation marks.
- Python allows either pair of single or double quotes.
- Subsets of strings can be taken using the slice operator (`[ ]` and `[:]`) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.
- The plus (`+`) sign is the string concatenation operator and the asterisk (`*`) is the repetition operator.

Python Strings (Cont.) - Lab

```
>>> str = 'Hello World!'
>>>
>>> print (str)                # Prints complete string
Hello World!
>>> print (str[0])             # Prints first character of the string
H
>>> print (str[2:5])           # Prints characters starting from 3rd to 5th
llo
>>> print (str[2:])            # Prints string starting from 3rd character
llo World!
>>> print (str * 2)            # Prints string two times
Hello World!Hello World!
>>> print (str + "TEST")       # Prints concatenated string
Hello World!TEST
>>>
```

Python Strings (Cont.) - Lab

```
1  name = 'Hello, World'
2  print(name.title())
3  print(name.upper())
4  print(name.lower())
5
6  first_name = 'michael'
7  last_name = 'jackson'
8  full_name = first_name + " " + last_name
9  print(full_name)
10 print("Hello, " + full_name.title() + "!")
11
12 print("Languages:\nPython\nC\nJavaScript")
13 print("Languages:\n\tPython\n\tC\n\tJavaScript")
14
```

Python Strings (Cont.) - Lab

```
1 favorite_language = 'python  '
2 print(favorite_language)
3
4 print(favorite_language.rstrip())
5 print(favorite_language)
6
7 favorite_language = favorite_language.rstrip()
8 print(favorite_language)
9
10 favorite_language = '  python  '
11 print(favorite_language.rstrip())
12
13 print(favorite_language.lstrip())
14
15 print(favorite_language.strip())
16
```

Python Strings (Cont.) - Lab

```
#age = 23
#message = "Happy " + age + "rd Birthday!"
#print(message)


age = 23
message = "Heppy " + str(age) + "rd Birthday!"
print(message)

```

Python Lists

- Lists are the most versatile of Python's compound data types.
- A list contains items separated by commas and enclosed within square brackets (`[]`).
- To some extent, lists are similar to arrays in C.
- One of the differences between them is that all the items belonging to a list can be of different data type.

Python Lists (Cont.)

- The values stored in a list can be accessed using the slice operator (`[ ]` and `[:]`) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (`+`) sign is the list concatenation operator, and the asterisk (`*`) is the repetition operator.

Python Lists (Cont.) - Lab

```
>>> list = ['abcd', 786, 2.23, 'john', 70.2]
>>> tinylist = [123, 'john']
>>>
>>> print (list)                          # Prints complete list
['abcd', 786, 2.23, 'john', 70.2]
>>> print (list[0])                       # Prints first element of the list
abcd
>>> print (list[1:3])                     # Prints elements starting from 2nd till 3rd
[786, 2.23]
>>> print (list[2:])                      # Prints elements starting from 3rd element
[2.23, 'john', 70.2]
>>> print (tinylist * 2)                  # Prints list two times
[123, 'john', 123, 'john']
>>> print (list + tinylist)               # Prints concatenated lists
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
>>>
```

Python Tuples

- A tuple is another sequence data type that is similar to the list.
- A tuple consists of a number of values separated by commas.
- Lists are enclosed in brackets (`[ ]`) and their elements and size can be changed.
- But, tuples are enclosed in parentheses (`( )`) and cannot be updated.
- Tuples can be thought of as read-only lists.

Python Tuples (Cont.) - Lab

```
>>> tuple = ('abcd', 786, 2.23, 'john', 70.2)
>>> tinytuple = (123, 'john')
>>>
>>> print (tuple)                # Prints complete tuple
('abcd', 786, 2.23, 'john', 70.2)
>>> print (tuple[0])             # Prints element of the tuple
abcd
>>> print (tuple[1:3])           # Prints elements starting from 2nd till 3rd
(786, 2.23)
>>> print (tuple[2:])            # Prints elements starting from 3rd element
(2.23, 'john', 70.2)
>>> print (tinytuple * 2)        # Prints tuple two times
(123, 'john', 123, 'john')
>>> print (tuple + tinytuple)  # Prints concatenated tuple
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
>>>
```

Python Dictionary

- Is kind of hash-table type.
- Works like associative arrays or hashes found in Perl and consist of key-value pairs.
- A dictionary key can be almost any Python type, but are usually numbers or strings.
- Values, on the other hand, can be any arbitrary Python object.
- Is enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

Python Dictionary (Cont.) - Lab

```
>>> dict = {}
>>> dict['one'] = "this is one"
>>> dict[2] = "This is two"
>>>
>>> tinydict = {'name' : 'john', 'code' : 6734, 'dept' : 'sales'}
>>>
>>> print (dict['one'])                    # Prints value for 'one' key
this is one
>>> print (dict[2])                        # Prints value for 2 key
This is two
>>> print (tinydict)                       # Prints complete dictionary
{'name': 'john', 'code': 6734, 'dept': 'sales'}
>>> print (tinydict.keys())                # Prints all the keys
dict_keys(['name', 'code', 'dept'])
>>> print (tinydict.values())              # Prints all the values
dict_values(['john', 6734, 'sales'])
>>>
```

Data Type Conversion

- Sometimes, may need to perform conversions between the built-in types.
- To convert between types, simply use the type-names as a function.
- There are several built-in functions to perform conversion from one data type to another.
- These functions return a new object representing the converted value.

Data Type Conversion (Cont.)

- `int(x [,base])`
  - Converts `x` to an integer.
  - The base specifies the base if `x` is a string.
- `float(x)`
  - Converts `x` to a floating-point number.
- `complex(real [, imag])`
  - Creates a complex number.
- `str(x)`
  - Converts object `x` to a string representation.

Data Type Conversion (Cont.)

- `repr(x)`
  - Converts object `x` to an expression string.
- `eval(str)`
  - Evaluates a string and returns an object.
- `tuple(s)`
  - Converts `s` to a tuple.
- `list(s)`
  - Converts `s` to a list.

Data Type Conversion (Cont.)

- `set(s)`
  - Converts `s` to a set.
- `dict(d)`
  - Creates a dictionary. `d` must be a sequence of (key, value) tuples.
- `frozenset(s)`
  - Converts `s` to a frozen set.
- `chr(x)`
  - Converts an integer to a character.

Data Type Conversion (Cont.)

- **`unichr(x)`**
  - Converts an integer to a Unicode character.
- **`ord(x)`**
  - Converts a single character to its integer value.
- **`hex(x)`**
  - Converts an integer to a hexadecimal string.
- **`oct(x)`**
  - Converts an integer to an octal string.

Data Type Conversion - Lab

```
>>>
>>> int('1234567890')
1234567890
>>>
>>> float('123.4567')
123.4567
>>>
>>> complex('1 + 2j')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    complex('1 + 2j')
ValueError: complex() arg is a malformed string
>>> complex('1+2j')
(1+2j)
>>>
```

Data Type Conversion - Lab

```
1  #input_test.py
2  print("첫 번째 숫자를 입력하세요 : ")
3  a = input()
4
5  print("두 번째 숫자를 입력하세요 : ")
6  b = input()
7
8  result = int(a) * int(b)
9
10 print("{0} * {1} = {2}".format(a, b, result))
11
```

Console

<terminated> Hello.py [/usr/bin/python3.6]

```
첫 번째 숫자를 입력하세요 :
5
두 번째 숫자를 입력하세요 :
4
5 * 4 = 20
```