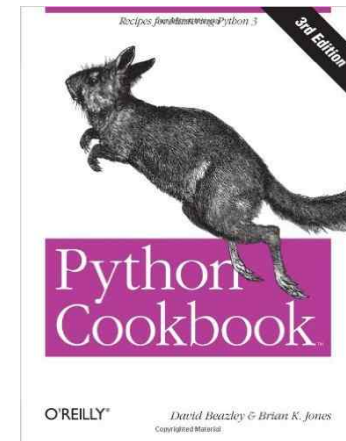
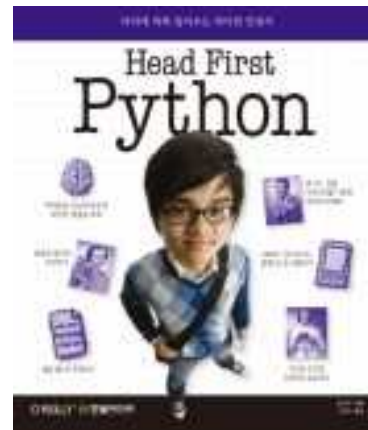
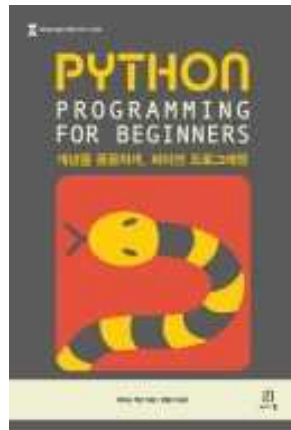
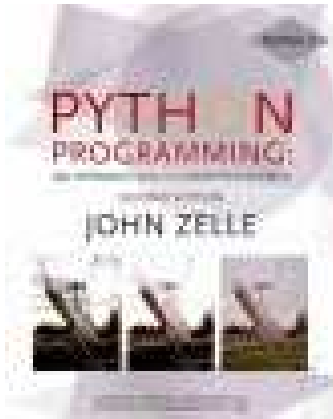


Python Tutorial

Special Thanks to “2015년 1학기 컴퓨터개념및 실습”의조교 유강민
박사과정

• Python Books



• Online Tutorials

<https://docs.python.org/2/tutorial/index.html>

<http://www.python-course.eu/index.php>

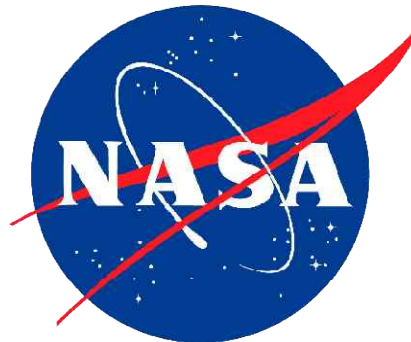
<http://interactivepython.org/courselib/static/thinkcspy/index.html>

• Just class notes + Googling

Why Python



- General-purpose, High-level, Scripting Language
- First appeared early 90s, invented by [Guido van Rossum](#)
- Easy to use, easy to learn
- Widely used as
 - Scientific libraries
 - Web Frameworks
 - Backend Frameworks
 - UI Frameworks
 - Graphic Frameworks
 - Data Mining Frameworks
 - And many others...



Why Python?: Advantages vs Disadvantages

- Advantages

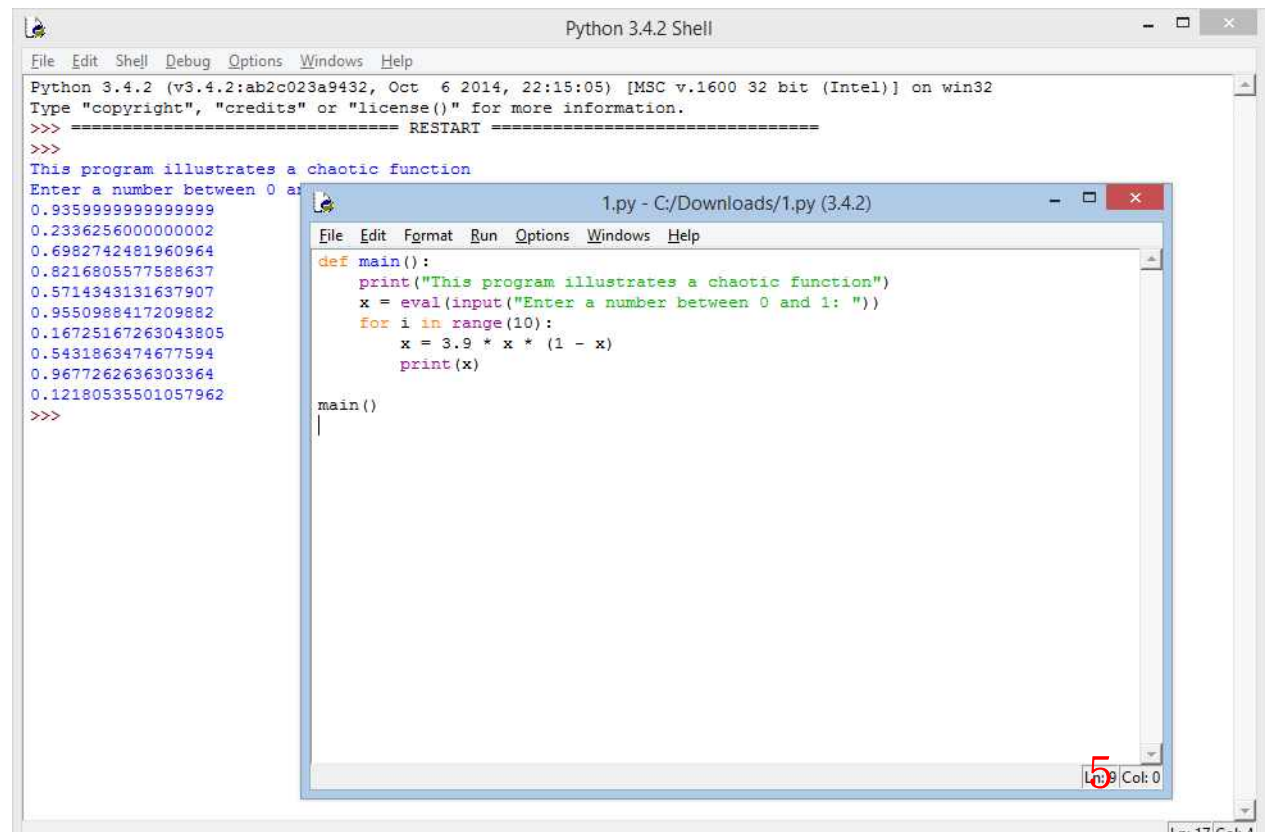
- Fast prototype testing
- Minimal development effort
- High readability

- Disadvantages

- As a scripting language, it requires a interpreter
- Performance might be an issue (memory, computation)
- Weak typing might be harder to debug

Ways to Use Python: Python IDLE

- Easy to use Interactive Development Environment (IDE)
- De-facto standard IDE for learning Python
- Provides simple debugging tool
- Provides simple code completion



The screenshot displays the Python 3.4.2 Shell and Editor windows. The Shell window shows the execution of a program that prints a list of numbers and prompts the user to enter a number between 0 and 1. The Editor window shows the source code for the program, which is a simple chaotic function.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct 6 2014, 22:15:05) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
This program illustrates a chaotic function
Enter a number between 0 and 1:
0.9359999999999999
0.23362560000000002
0.6982742481960964
0.8216805577588637
0.5714343131637907
0.9550988417209882
0.16725167263043805
0.5431863474677594
0.9677262636303364
0.12180535501057962
>>>
```

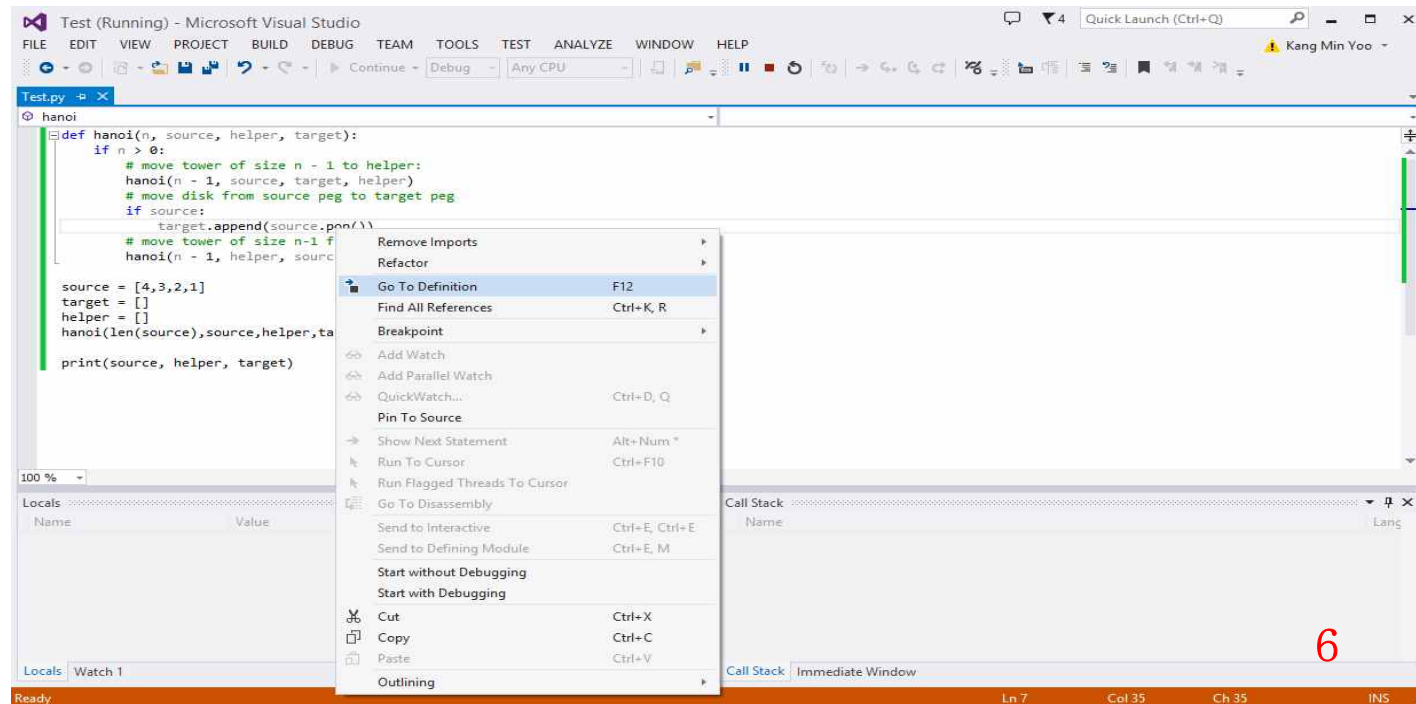
```
1.py - C:/Downloads/1.py (3.4.2)
File Edit Format Run Options Windows Help
def main():
    print("This program illustrates a chaotic function")
    x = eval(input("Enter a number between 0 and 1: "))
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)

main()
|
```

Ways to Use Python: Python Tools for Visual Studio

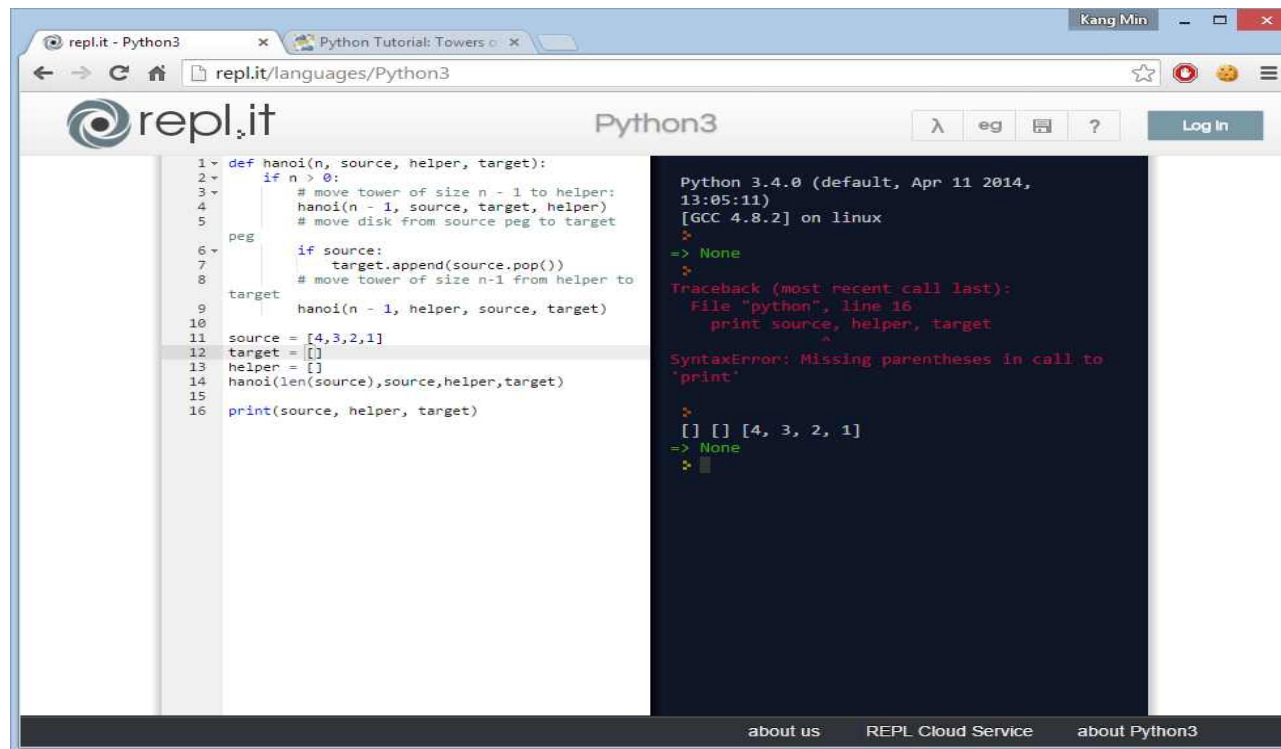
<https://www.visualstudio.com/en-us/features/python-vs.aspx>

- Has a steep learning curve, but very useful if used right
- Might be difficult for beginners in programming
- Supports most visual studio features
 - Finding references // Code completion // Syntax checking
 - Simple semantics checking // Full stack Debugging
 - Inspection // And many others...



Ways to Use Python

- Repl.it <http://repl.it/>
 - Surprisingly good and very easy to use
 - Requires no installation of the interpreter on the machine
 - Can be used interactively
 - However, only Python 3.4.0 is available
 - The latest Python version is 3.4.3
 - Scripts might be interpreted differently



The screenshot shows the Repl.it web interface for Python3. The left pane contains a Python script for the Hanoi Tower problem. The right pane shows the execution output, which includes the Python version (3.4.0), GCC version (4.8.2), and a traceback error message: 'SyntaxError: Missing parentheses in call to \'print\''. The error points to line 16 of the script, where the print statement is: `print(source, helper, target)`. The output also shows the initial state of the source, target, and helper lists: `[4, 3, 2, 1]`, `[]`, and `[]` respectively.


```
1~ def hanoi(n, source, helper, target):
2~     if n > 0:
3~         # move tower of size n - 1 to helper:
4~         hanoi(n - 1, source, target, helper)
5~         # move disk from source peg to target
6~     peg = source
7~     if source:
8~         target.append(source.pop())
9~     # move tower of size n-1 from helper to
10~    target
11~    hanoi(n - 1, helper, source, target)
12~
13~ source = [4,3,2,1]
14~ target = []
15~ helper = []
16~ hanoi(len(source),source,helper,target)
17~ print(source, helper, target)
```

Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
=> None
Traceback (most recent call last):
 File "python", line 16,
 print source, helper, target
SyntaxError: Missing parentheses in call to
'print'
[4, 3, 2, 1]
=> None

Ways to Use Python

- Repl.it
 - Fast
 - Portable
 - Suitable for prototype testing
- Python IDLE
 - Readily available in the official python install package
 - Fairly easy to use
 - Features debugging
- Python Tools for Visual Studio
 - Contains the complete feature for programmers
 - The learning curve might be steep
 - Debugging, Refactoring, Syntax Checking, Syntax Highlighting, Dependency Management, and many more...

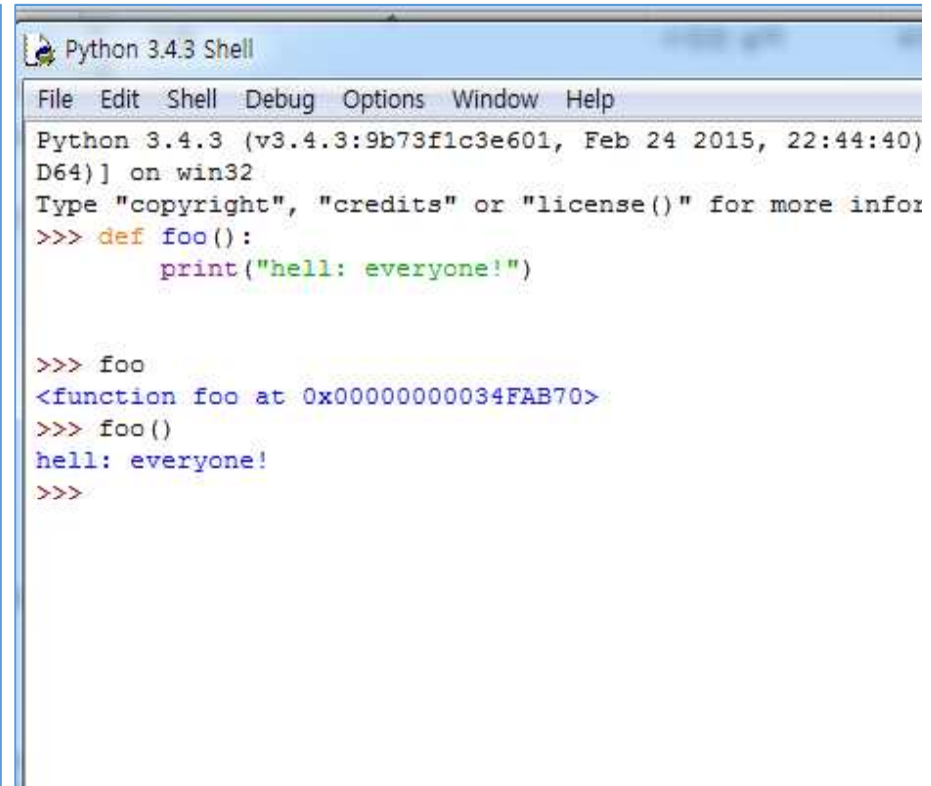
IDLE: Initial Screen of IDLE



The screenshot shows the Python 3.4.3 Shell window. The title bar reads "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the following text:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MS  
D64)] on win32  
Type "copyright", "credits" or "license()" for more informati  
>>> |
```

Initial Coding in IDLE

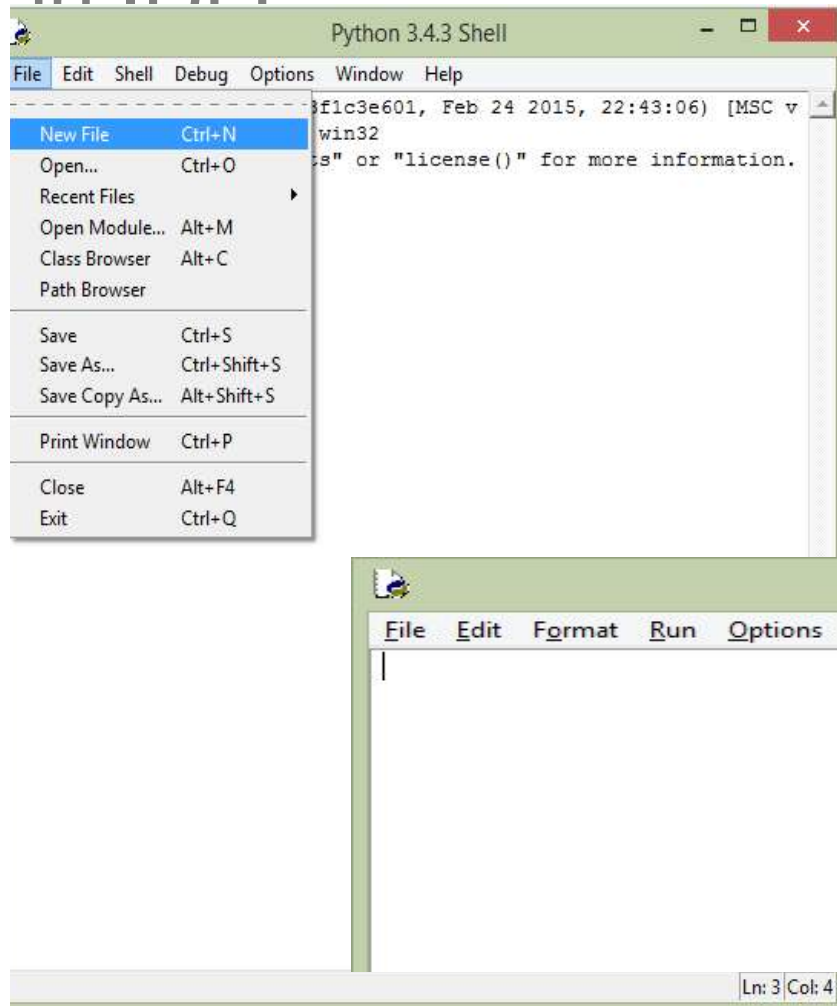


The screenshot shows the Python 3.4.3 Shell window with the same title bar and menu bar as the previous image. The main text area displays the following code and output:

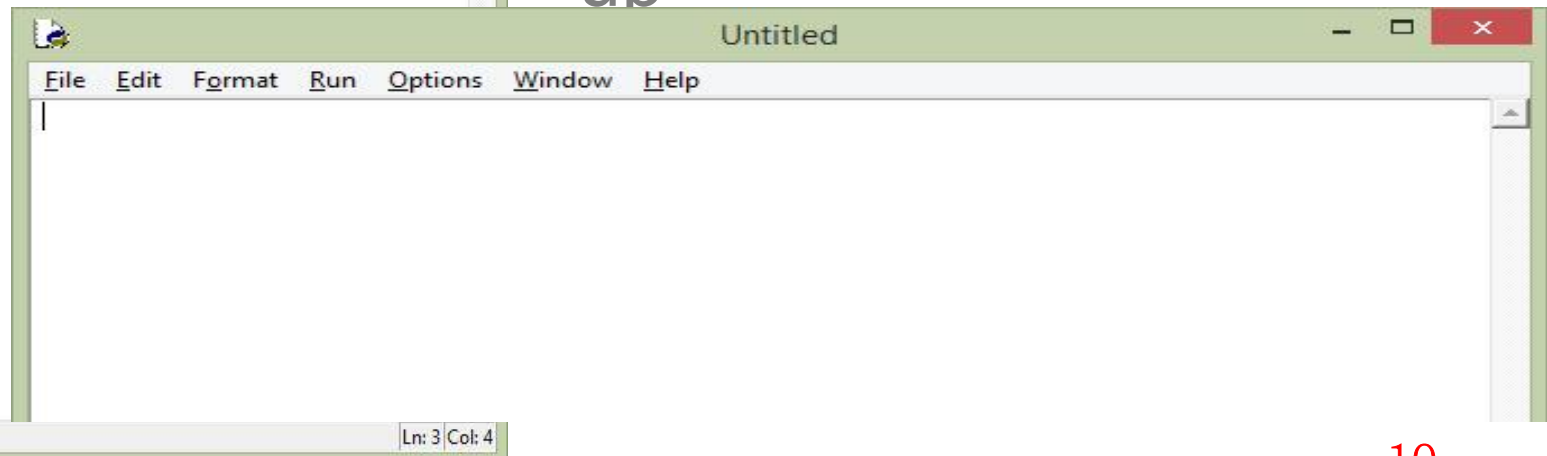
```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40)  
D64)] on win32  
Type "copyright", "credits" or "license()" for more infor  
>>> def foo():  
    print("hell: everyone!")  
  
>>> foo  
<function foo at 0x00000000034FAB70>  
>>> foo()  
hell: everyone!  
>>>
```

Suppose you finish up coding into IDLE and you want to save your Python code in your directory!

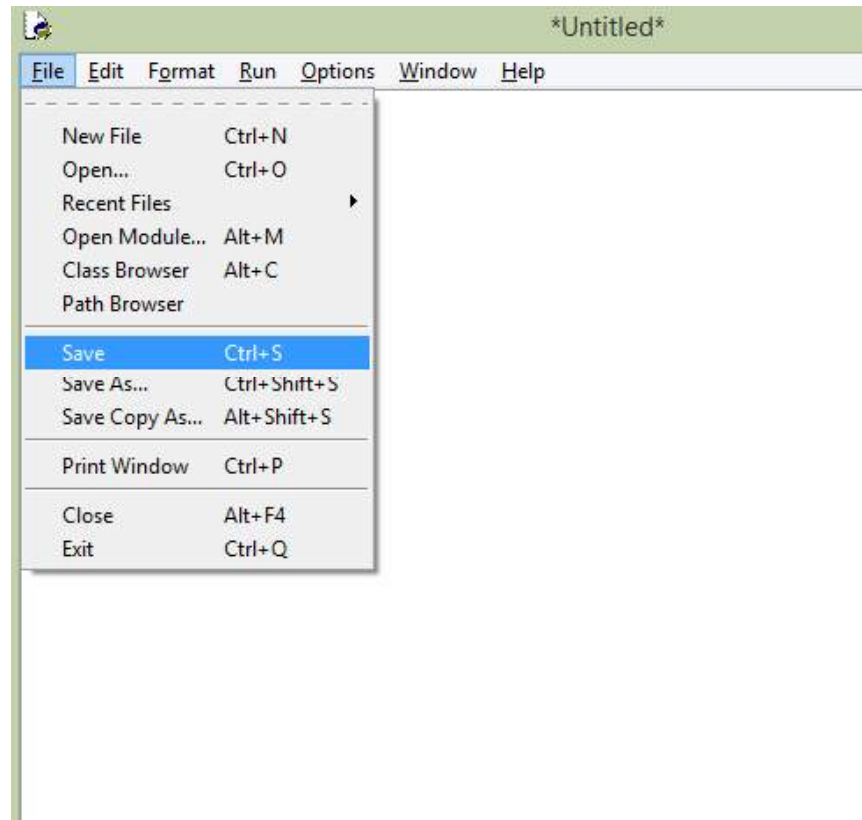
Creating a new script file in IDLE



A new “untitled” window for a new script is popped up

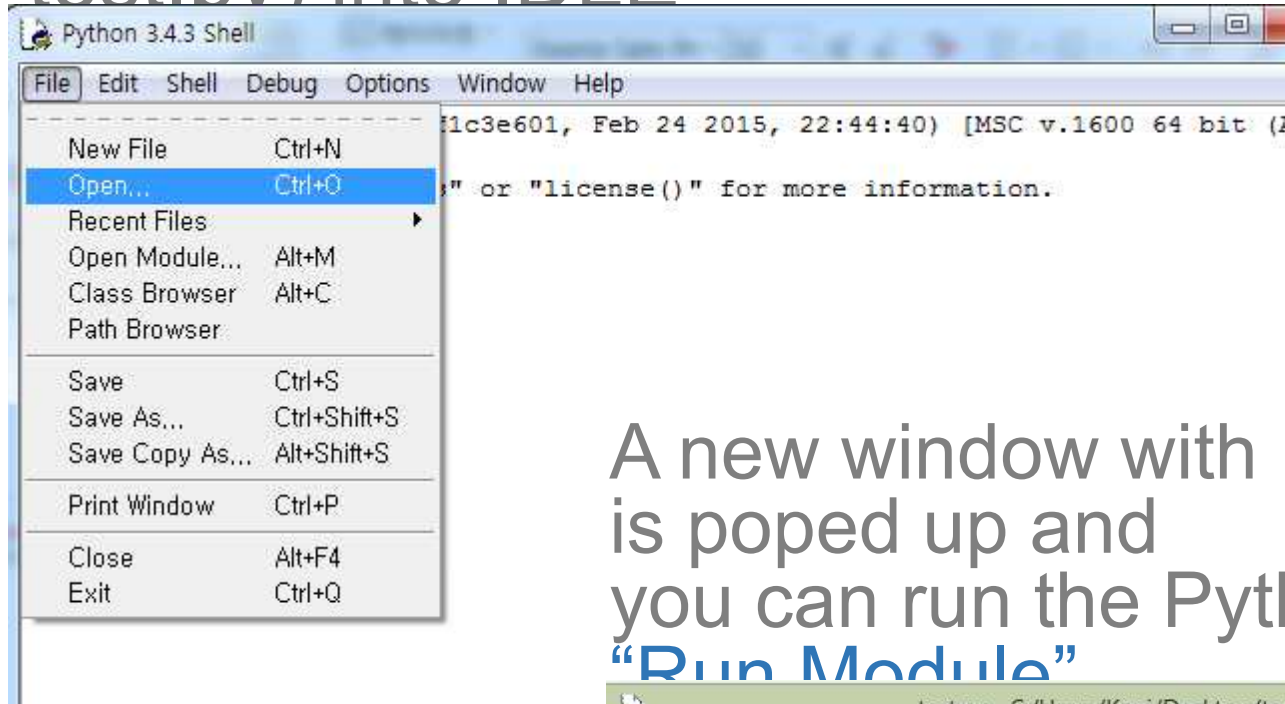


- Cut & Paste Python codes in IDLE window to “untitled” window
- Then, save the code as a new Python file (say, `test.py`)

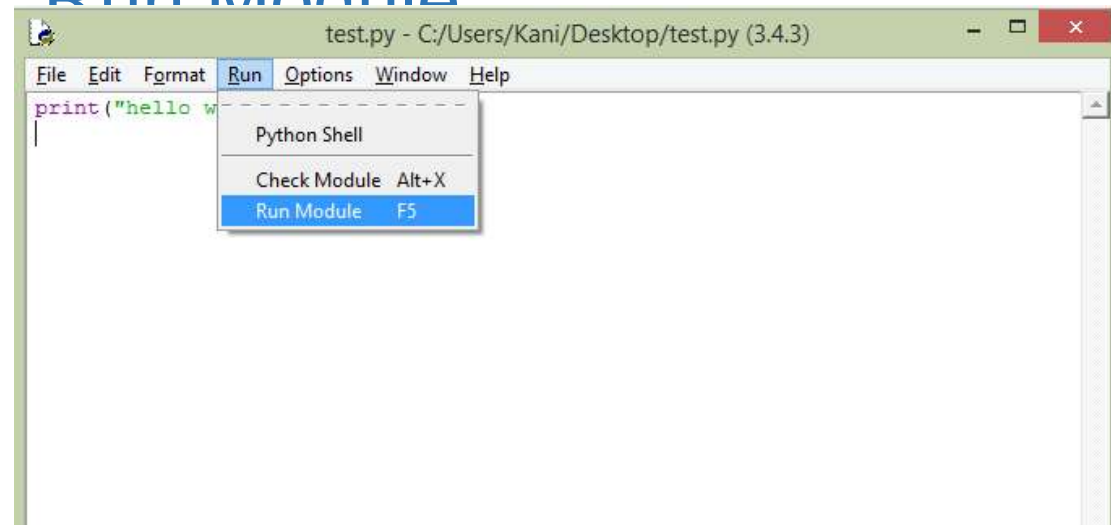


- Now you have “`test.py`” in your directory

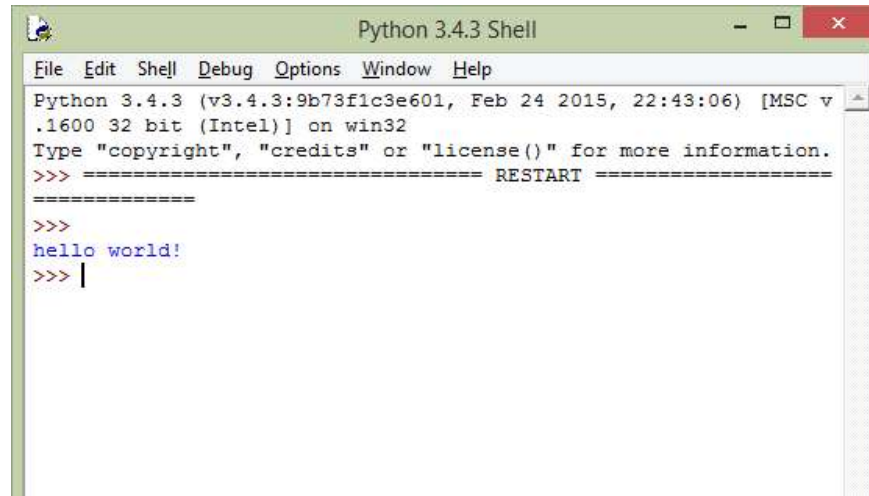
If you want to read an existing Python file (say, test.py) into IDLE



A new window with Python file name is popped up and you can run the Python file by clicking “Run Module”



Test results are displayed in a new (existing) shell window



The image shows a screenshot of a 'Python 3.4.3 Shell' window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area displays the following content:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v
.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> hello world!
>>> |
```

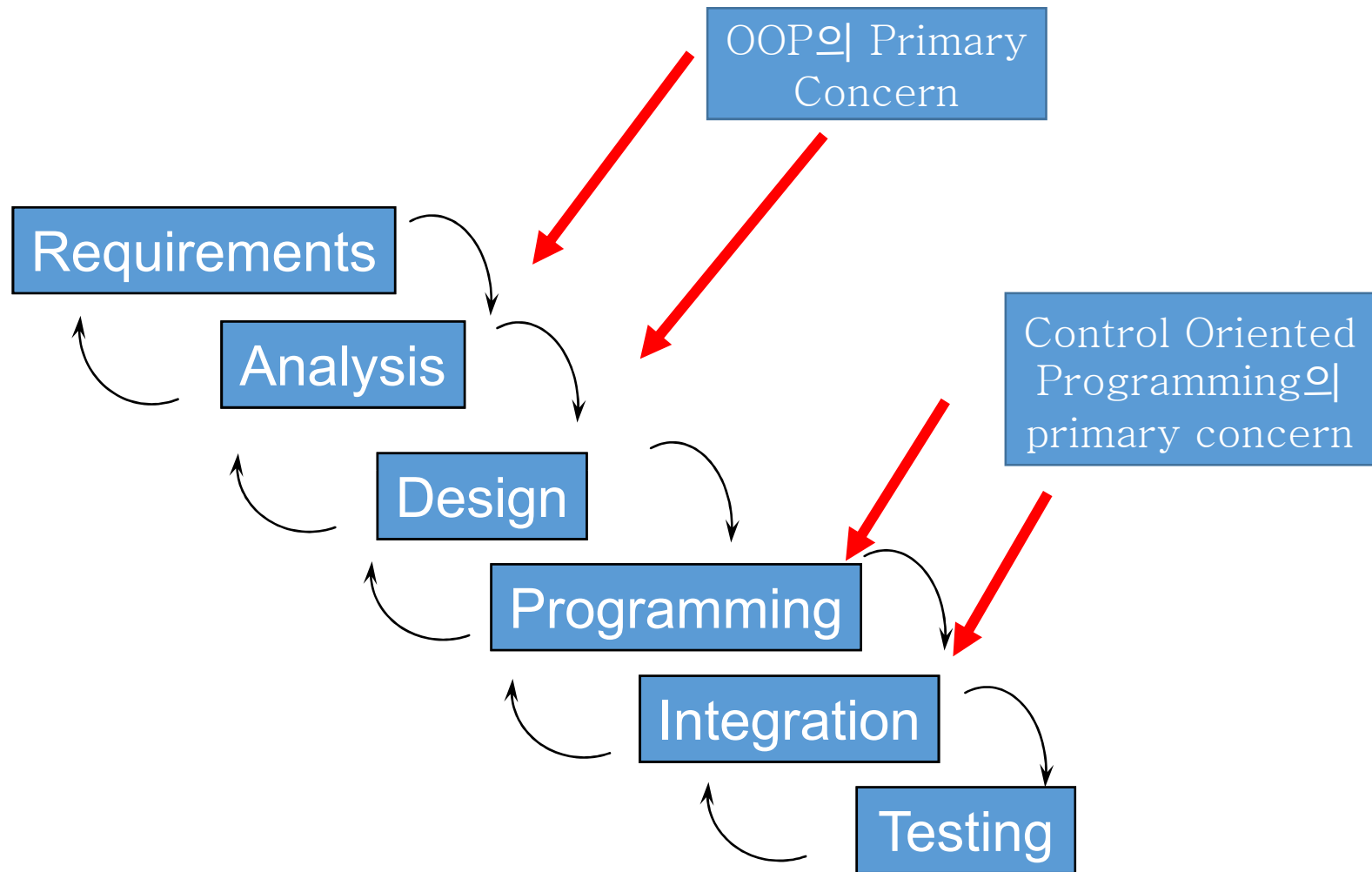
Programming Languages: Compiler vs Interpreter

- **Compiled programs** generally run **faster** since the translation of the source code happens only once.
- Once program is compiled, it can be executed over and over without the source code or compiler.
- **Interpreted programs** are more **portable**, meaning the same program can run on a Intel PC and on a Mac as long as the interpreter is available
- Interpreted languages are part of a more **flexible** programming environment since they can be developed and run interactively

The Software Development Process: The WaterFall Model

- Analyze the Problem
 - Figure out exactly the problem to be solved.
- Determine Specifications
 - Describe exactly what your program will do. (not **How**, but **What**)
 - Includes describing the inputs, outputs, and how they relate to one another.
- Create a Design
 - Formulate the overall structure of the program. (**how** of the program gets worked out)
 - You choose or develop your own algorithm that meets the specifications.
- Implement the Design (coding!)
 - Translate the design into a computer language.
- Test/Debug the Program
 - Try out your program to see if it worked.
 - Errors (Bugs) need to be located and fixed. This process is called **debugging**.
 - Your goal is to find errors, so try everything that might “break” your program!
- Maintain the Program
 - Continue developing the program in response to the needs of your users.
 - **In the real world**, most programs are never completely finished – **they evolve over time**.

Waterfall SW Development Model



Python Features

Temperature Converter: Example Program

- Problem Analysis

- the temperature is given in Celsius, user wants it expressed in degrees Fahrenheit.

- Program Specification

- Input – temperature in Celsius
 - Output – temperature in Fahrenheit: $\frac{9}{5}(\text{input}) + 32$

```
def main():  
    celsius = eval(input("What is the Celsius temperature? "))  
    fahrenheit = (9/5) * celsius + 32  
    print("The temperature is ", fahrenheit, " degrees Fahrenheit.")  
  
main()
```

Temperature Converter: Testing

```
>>>  
What is the Celsius temperature? 0  
The temperature is 32.0 degrees Fahrenheit.  
>>> main()  
What is the Celsius temperature? 100  
The temperature is 212.0 degrees Fahrenheit.  
>>> main()  
What is the Celsius temperature? -40  
The temperature is -40.0 degrees Fahrenheit.  
>>>
```

Python Syntax

- Functions are defined by “def” keyword
- No braces “{}” for scopes unlike C
- Purple keywords (highlighted in IDLE) are

Python’s built-in operators

- Other examples include return

```
def main():  
    celsius = eval(input("What is the Celsius temperature? "))  
    fahrenheit = (9/5) * celsius + 32  
    print("The temperature is ", fahrenheit, " degrees Fahrenheit.")  
  
main()
```

Assigning Input

- The purpose of an input statement is to get input from the user and store it into a variable.

```
<variable> = eval(input(<prompt>))
```

- First the prompt is printed
- The input part waits for the user to enter a value and press <enter>
- The expression that was entered is evaluated to turn it from a string of characters into a Python value (a number).
- The value is assigned to the variable.

- ```
>> x = input("type your value:")
```

- ```
>> x = eval(input("type your value:"))
```

Elements of Programs: Identifiers

- Names

- Names are given to variables (celsius, fahrenheit), modules (main, convert), etc.
- These names are called **identifiers**
- Every identifier must begin with a letter or underscore (“_”), followed by any sequence of letters, digits, or underscores.
- Identifiers are **case sensitive**.

- These are all different, valid names

- X Celsius fahrenheit
- Spam spam spAm
- Spam_and_Eggs Spam_And_Eggs

Elements of Programs: Identifiers

• Reserved Words

- Some identifiers are part of Python itself. These identifiers are known as **reserved words** or **keywords**.
- This means they are **not available for you** to use as a name for a variable, etc. in your program.

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

Table 2.1: Python Reserved Words.

Elements of Programs: Expressions

- The fragments of code that produce or calculate new data values are called **expressions**.
- **Literals** are used to represent a specific value, e.g. 3.9, 1, 1.0
- Simple identifiers can also be expressions.

```
>>> x = 5
>>> x
5
>>> print(x)
5
```


Elements of Programs: Output Statements

- A print statement can print any number of expressions.
- Successive print statements will display on separate lines.
- A bare print will print a blank line.

- | | |
|--|-----------------|
| · <code>print(3+4)</code> | 7 |
| · <code>print(3, 4, 3+4)</code> | 3, 4, 7 |
| · <code>print()</code> | |
| · <code>print("The answer is", 3+4)</code> | The answer is 7 |

STRINGS: **SPECIAL CHARACTERS**

"\n" = **newline** character
(enter key)

`print "test\n\ntest"`

test

test

"\t" = **TAB** character
(tab key)

`print "test\t\t test"`

test

test

```
'te\n\nst'
>>> print x
te

st
>>> x = "te\n\nst"
>>> x
'te\n\nst'
>>> print x
te

st
>>> print "test"
test
>>> x = "test\t\ttest"
>>> print x
test          test
>>>
```

Elements of Programs: Assignment Statements

- Simple Assignment
 - $\langle \text{variable} \rangle = \langle \text{expr} \rangle$
 - $\langle \text{variable} \rangle$ is an identifier, $\langle \text{expr} \rangle$ is an expression
- The expression on the RHS is evaluated to produce a value which is then associated with the variable named on the LHS.

`x = 3.9 * x * (1-x)`

`fahrenheit = 9/5 * celsius + 32`

`x = 5`

Elements of Programs: Simultaneous Assignment

- Several values can be calculated at the same time
 - `<var>, <var>, ... = <expr>, <expr>, ...`
 - Evaluate the expressions in the RHS and assign them to the variables on the LHS

```
sum, diff = x+y, x-y
```

- How could you use this to swap the values for x and y?
 - Would this work?
`x = y`
`y = x`
- We could use a temporary variable...
- Or We can swap the values of two variables quite [easily in Python!](#)

```
x, y = y, x
>>> x = 3
>>> y = 4
>>> print x, y
3 4
>>> x, y = y, x
>>> print x, y
4 3
```

Elements of Programs: Definite Loops

- A *definite* loop executes a definite number of times,
 - i.e., at the time Python starts the loop it knows exactly how many *iterations* to do.

```
for <var> in <sequence>:  
    <body>
```

- The beginning and end of the body are indicated by indentation.
- Examples

```
>>> for i in [0,1,2,3]:  
    print (i)  
  
0  
1  
2  
3
```

```
>>> for odd in [1, 3, 5, 7]:  
    print(odd*odd)  
  
1  
9  
25  
49  
  
>>>
```

Elements of Programs: Definite Loops

- Sequences can be created by `range` keyword.

```
>>> list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
for i in range(4):  
is equivalent to  
for i in [0, 1, 2, 3]:
```

```
>>> answer = []  
    for i in range(1,10):  
        answer.append(str(i))
```

참고: `answer = answer.append(str(i))` 는 error!
왜냐하면 `append`는 `answer`에 `append`하는것이고 `return`은 없으므로

Numeric Data Types

- Types
 - Integers (`int`) – whole numbers
 - E.g. 3, 5
 - Floating point values (`float`) – with decimal point
 - E.g. 3.1, 5.1, 6.
- Types can be probed using “type” built-in function

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
```

Numeric Data Types: Operations

- Operations on ints produce ints, operations on floats produce floats (except for /).

```
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3.3333333333333335
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
>>> 3.0 + 4
7.0
>>> 3.0+4.0
7.0
>>> 3.0*4.0
12.0
>>> 3*4
12
```


Python Built-in Functions

		Built-in Functions		
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

Using Libraries (or Modules)

- Modules contain a set of useful functions or classes or even routines
- Some additional standard modules like “`math`”, “`time`” or “`datetime`” etc. can be imported using keyword “`import`”
- Standard modules are listed here.

(<https://docs.python.org/3/py-modindex.html>)

- Ex)
 - `>> import math`
 - `>> math.sqrt(3)`
 - `1.7320508075688772`
 - `>> - b + math.sqrt(b * b - 4 * a * c) / (2 * a)`
- As always, “`help(<module name>)`” can give you lots of information

Factorial Program

```
# factorial.py
#   Program to compute the factorial of a number
#   Illustrates for loop with an accumulator
def main():

    n = eval(input("Please enter a whole number: "))

    fact = 1
    for factor in range(n,1,-1):
        fact = fact * factor

    print("The factorial of", n, "is", fact)
main()
```

Factorial Program

- Interesting thing to note is that Python **expands** integers into biginteger automatically in newer versions of Python

```
>>> main()
```

Please enter a whole number: 100

The factorial of 100 is

9332621544394415268169923885626670049071596826438162146859
2963895217599993229915608941463976156518286253697920827223
7582511852109168640000000000000000000000000000

- Python has **built-in** support for integers exceeding 32-bit or 64-bit

Type Conversions

```
>>> float(22//5)
```

```
4.0
```

```
>>> float(3)
```

```
3.0
```

```
>>> float(3.3)
```

```
3.3
```

```
>>> int(4.5)
```

```
4
```

```
>>> int(4.9)
```

```
4
```

```
>>> int(4.1)
```

```
4
```

```
>>> int(4.999999999999)
```

```
4
```

```
>>> int(4)
```

```
4
```

```
>>> round(3.9)
```

```
4
```

```
>>> round(3)
```

```
3
```

```
>>> round(3.5)
```

```
4
```

```
>>> str(8)
```

```
'8'
```

```
>>> 32/32
1
>>> 3/2
1
>>> 100.0000000
100.0
>>> 100.0
100.0
>>> 100
100
>>> 3/2.0
1.5
>>> 3/2
1
```

```
>>> 100.0000000
100.0
>>> 100.0
100.0
>>> 100
100
>>> 3/2.0
1.5
>>> float(3/2)
1.0
>>> float(3)/2
1.5
>>> float(5)/4
1.25
>>> int(3.0)
3
```

```
>>> 5 % 4
1
>>> 10 % 5
0
>>> 11% 5
1
>>> 12 %5
2
>>> float(5)/4
1.25
>>> int(3.0)
3
>>> 5 % 4
1
>>> 10 % 5
0
```

```
>>> 11% 5
1
>>> 12 %5
2
>>> 12**2
144
>>> 16**0.5
4.0
>>> 24**0.5
4.898979485566356
>>> 5 + (35 8 13 + 2 )
SyntaxError: invalid syntax
>>> 5 + (35 + 13 + 2 )
55
>>> 5*(5+5)
50
```

```

Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MS
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
tion.
>>> x = 'ham'
>>> x
'ham'
>>> x = "hamsandwich"
>>> x
'hamsandwich'
>>> y = x + "book"
>>> y
'hamsandwichbook'
>>> y = x + " book"
>>> y
'hamsandwich book'
>>>

```

```

>>> x = "hamsandwich"
>>> z = 10
>>> y = x + z

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    y = x + z
TypeError: cannot concatenate 'str' and 'int' objects
>>> y = x + str(z)
>>> y
'hamsandwich10'
>>> y = "something %d" %z
>>> y
'something 10'
>>> y = "something %f" %z
>>> y
'something 10.000000'
>>> y = "Something %.3f" %z
>>> y
'Something 10.000'
>>> z = 1.6546546548
>>> y = "something %.3f" %z
>>> y
'something 1.655'

```

Sequences: String and Its Indexing

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```


Sequences: String and Its Indexing

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of n characters, the last character is at position $n-1$ since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
```

```
'b'
```

```
>>> greet[-3]
```

```
'B'
```

Sequences: String and *Substring*

- Slicing: `<string>[<start>:<end>]`
 - start and end should both be ints
 - contains the substring beginning at position start and runs up to but doesn't include the position end

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
'Hel'
```

```
>>> greet[5:9]
' Bob'
```

```
>>> greet[:5]
'Hello'
```

```
>>> greet[5:]
' Bob'
```

```
>>> greet[:]
'Hello Bob'
```

```
>>> greet[0:-3]
```

```
'Hello '
```

```
>>> greet[:-1]
```

```
'Hello Bo'
```

Sequences: String and Its Operators

- Concatenation (+)

```
>> "a" + "b"
```

```
'ab'
```

- Reptition (*)

```
>> "a" * 3
```

```
'aaa'
```

- Length (len)

```
>> len("a" * 3)
```

```
3
```

String: Examples

```
# get user's first and last names
first = input("Please enter your first name (all lowercase): ")
last = input("Please enter your last name (all lowercase): ")

# concatenate first initial with 7 chars of last name
print ("your_name = ", first[0] + ". " + last[:7])
```

```
>>>
```

```
Please enter your first name (all lowercase): john
```

```
Please enter your last name (all lowercase): doe
```

```
your_name = j. doe
```

Sequences: Lists

- Lists are a special kind of *sequence*, so sequence operations also apply to lists!

```
>>> [1,2] + [3,4]  
[1, 2, 3, 4]
```

```
>>> [1,2]*3  
[1, 2, 1, 2, 1, 2]
```

```
>>> grades = ['A', 'B', 'C', 'D', 'F']  
>>> grades[0]  
'A'  
>>> grades[2:4]  
['C', 'D']  
>>> len(grades)  
5
```

Sequence: Lists

- Strings are always sequences of characters
univ_name = "Seoul National Univ"
- but *lists* can be sequences of arbitrary values.
 - Lists can have numbers, strings, or both!

```
myList = [1, "Spam ", 4, "U"]
```

Lists: Example

```
# month2.py
# A program to print the month name, given it's number.
# This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = eval(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")

main()
```

Sequence. Lists are *mutable* (i.e. they can be changed).

- Parts of Strings can not be changed using operations.
 - As a side note, *Immutable* Lists are called Tuples

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"      #This is not allowed
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```


TUPLES

Just like Lists but
UNADJUSTABLE

x = ()
x = ("ham", 4, 5)

```
>>> x = ("ham", 4, 5)
>>> x
('ham', 4, 5)
>>> | I
```

No add/drop a part inside a tuple!

```
>>> x[2] = 8 # not allowed
```

Whole replacement is fine!

```
>>> x = ("egg", 7, 9, 10)
```

TUPLES VS. LISTS

More memory
efficient

Takes more
memory

Cannot be
adjusted

Adjustable

Sequence: Characters

- `ord()` : returns the numeric (ordinal) code of a single character.
- `chr()` : converts a numeric code to the corresponding character.

```
>>> ord("A")    # argument character의 ascii code에서의 위치
65
>>> ord("a")
97
>>> chr(97)      # 0 ~ 255를 받아서 ascii code를 return
'a'
>>> chr(65)
'A'
```

UTF-8 code (가변형 4 bytes)을 사용했다 하더라도 처음 1byte는 ASCII code와 동일함

Sequence: Strings

- How do we get the sequence of numbers to decode?
 - Read the input as a single string, then split it apart into substrings, each of which represents one number.
- The string class has a set of methods.
- *split() method*
 - splits the given string into substrings based on spaces.

```
>>> a = "Hello string methods!"
>>> a.split()
['Hello', 'string', 'methods!']
>>> "Hello string methods!".split()
['Hello', 'string', 'methods!']
```

`a = a.split()` 를 써도 OK, `a.split()`은 리스트를 return하므로!
But, `a = a.append("dd")`를 하면 `a`에는 nothing !

Sequence: String Splitting function

- Split can be used on characters other than space, by supplying the character as a parameter.

```
>>> "32,24,25,57".split(",")  
['32', '24', '25', '57']  
>>> "abcPdefPghi".split("P")  
['abc', 'def', 'ghi']
```

```
>>> list("CMU")  
['C', 'M', 'U']
```

Sequence: String operators

- `s.capitalize()`
- `s.title()`
- `s.center(width)`
- `s.count(sub)` – Count # of occurrences of sub in s
- `s.find(sub)` – Find first pos where sub occurs in s
- `s.join(list)` – Concatenate list of strings into one string using s as separator.
- `s.ljust(width)`
- `s.lower()` – Copy of s in all lowercase letters
- `s.lstrip()` – Copy of s with leading whitespace removed
- `s.replace(oldsub, newsub)` – Replace occurrences of oldsub in s with newsub
- `s.rfind(sub)` – Like find, but returns the right-most position
- `s.rjust(width)` – Like ljust
- `s.rstrip()` – Like lstrip
- `s.split()`

```

Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MS
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
tion.
>>> x = 'ham'
>>> x
'ham'
>>> x = "hamsandwich"
>>> x
'hamsandwich'
>>> y = x + "book"
>>> y
'hamsandwichbook'
>>> y = x + " book"
>>> y
'hamsandwich book'
>>>

```

```

>>> x = "hamsandwich"
>>> z = 10
>>> y = x + z

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    y = x + z
TypeError: cannot concatenate 'str' and 'int' objects
>>> y = x + str(z)
>>> y
'hamsandwich10'
>>> y = "something %d" %z
>>> y
'something 10'
>>> y = "something %f" %z
>>> y
'something 10.000000'
>>> y = "Something %.3f" %z
>>> y
'Something 10.000'
>>> z = 1.6546546548
>>> y = "something %.3f" %z
>>> y
'something 1.655'

```

```

>>> "ham" in "hamsandwich"
True
>>> 'a' in 'ham'
True
>>> x = []
>>> x
[]
>>> x = ["ham", 4, 2.2]
>>> x
['ham', 4, 2.2]
>>> x.append(5)
>>> x
['ham', 4, 2.2, 5]
>>> x.insert(1, 3.1415)
>>> x
['ham', 3.1415, 4, 2.2, 5]

```

```

>>> x.pop(1)
3.1415
>>> x
['ham', 4, 2.2, 5]
>>> len("words")
5
>>> len(x)
4
>>> |

```

```

>>> list("ham")
['h', 'a', 'm']
>>> x = "ham"
>>> y = list(x)
>>> y
['h', 'a', 'm']
>>> y.append(x)
>>> y
['h', 'a', 'm', 'ham']
>>> list("ham")
['h', 'a', 'm']
>>> y = []
>>> y.append("ham")
>>> y
['ham']

```

```

>>> "s"
's'
>>> "s" in "something"
True
>>> "s" in y
False

```

```

>> x = ['a', 'b', 'c']      >> x = ['a', 'b', 'c']
>> x = x.pop(1)            >> x.pop(1)
>> x                        >> x

```

```

y = y.append(x)  # 절대 안됨
y = list(y)     # 가능함

```

Boolean Expressions

- There is additional type called *bool* – it's either True or False

```
>> type(True)
```

```
<class 'bool'>
```

- Boolean expressions are always evaluated to True or False
- Format: `<expr> <relop> <expr>`

Python	Mathematics	Meaning
<	<	Less than
<=	≤	Less than or equal to
==	=	Equal to
>=	≥	Greater than or equal to
>	>	Greater than
!=	≠	Not equal to

Boolean Expressions. Comparisons by Types

- Operands in comparison operations should be compatible types
 - However, test for equality can be carried out with different types (but returns “False”)
- When comparing strings, the ordering is lexicographic

```
>> “aaa” > “abb”
```

```
False
```

```
>> 5 < 2.5
```

```
False
```

```
>> “a” < 572.0
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#13>", line 1, in <module>
```

```
    “a” < 572.0
```

```
TypeError: unorderable types: str() < int()
```

```
>> “a” == 572.0
```

Boolean Expressions

- The following compound comparisons are valid expressions in Python

E.g.

```
>> 1 < 5 < 7
```

```
True
```

```
>> 2 > 1 < 7
```

```
True
```

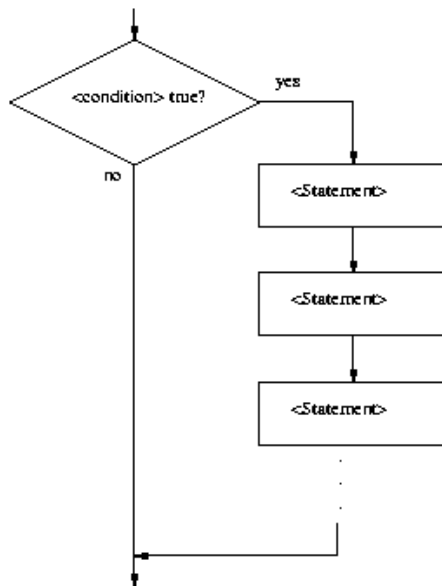
```
>> 5 > 4 > 3.2 >= 1 == 1 != 8
```

```
True
```

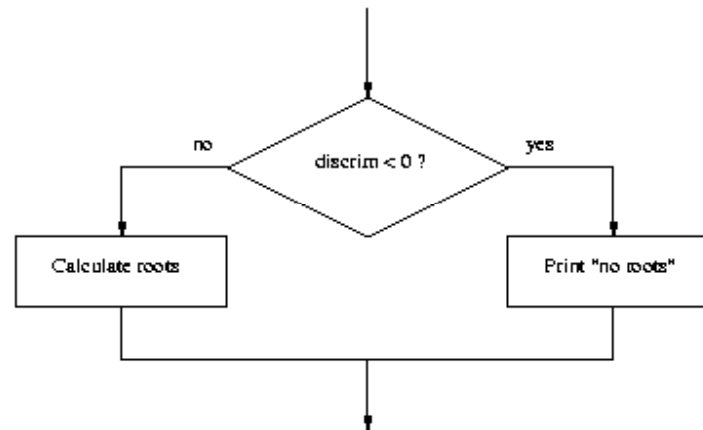
```
If 0 <= number <= 100 :  
    print(number)          # 이것도 가능
```

Decision Structures

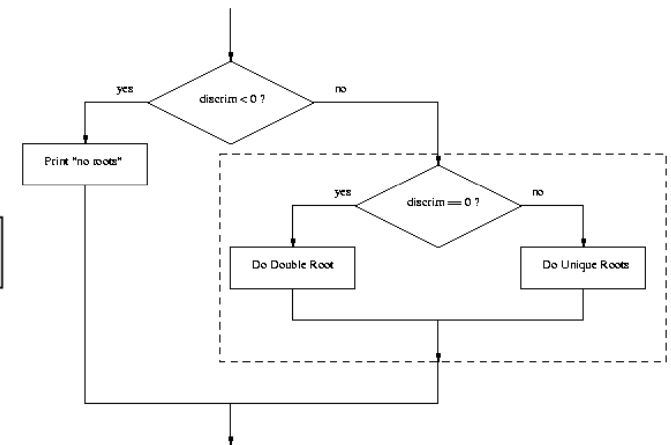
```
if <condition>:  
    <statements>
```



```
if <condition>:  
    <statements>  
else:  
    <statements>
```



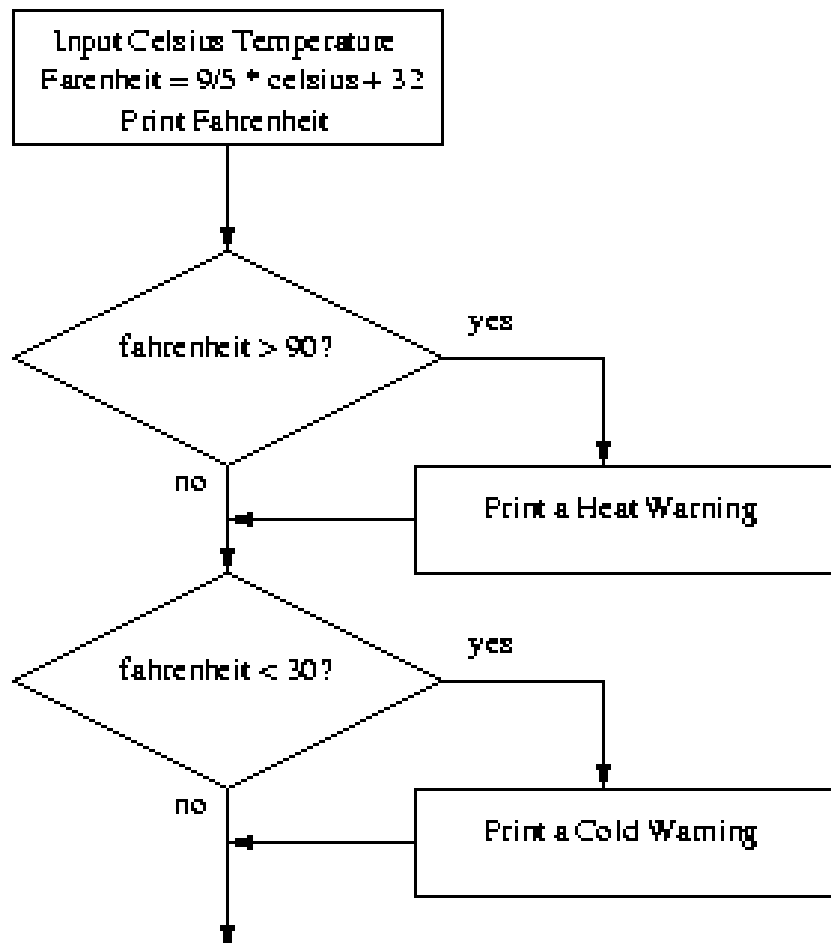
```
if <condition>:  
    <statements>  
elif <condition>:  
    <statements>  
else:  
    <statements>
```



One-way Decision Structure: Example

- Temperature Warnings

- Let's say we want to modify that program to print a warning when the weather is extreme.



One-way Decision Structure: Example

```
# convert2.py
#       A program to convert Celsius temps to Fahrenheit.
#       This version issues heat and cold warnings.

def main():
    celsius = eval(input("What is the Celsius temperature? "))
    fahrenheit = 9 / 5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees fahrenheit.")
    if fahrenheit >= 90:
        print("It's really hot out there, be careful!")
    if fahrenheit <= 30:
        print("Brrrrrr. Be sure to dress warmly")

main()
```

Predefined Attributes

- Called “special variables” or “magic variables”
 - They contain meta-data about script files / modules
- These variables have the form of `__<variable>__`, which is enclosed by two underscores
- One important variable is `__name__`
 - it tells us the **name** of the module
 - currently running script file will have `__name__ = “__main__”`

```
>> import math
>> math.__name__
'math'
>> __name__
'__main__'
```

- The complete list of predefined attributes are listed in <https://docs.python.org/2/reference/datamodel.html>
- `__name__`, `__dict__`, `__doc__`, `__code__`, 등등

Two-Way Decision Structure: Example

```
# quadratic3.py
#   A program that computes the real roots of a quadratic equation.
#   Illustrates use of a two-way decision

import math

def main():
    print "This program finds the real solutions to a quadratic\n"

    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print ("\nThe solutions are:", root1, root2 )

main()
```

Multi-Way Decision Structure: Example

```
# quadratic4.py
#   Illustrates use of a multi-way decision

import math

def main():
    print("This program finds the real solutions to a quadratic\n")
    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    elif discrim == 0:
        root = -b / (2 * a)
        print("\nThere is a double root at", root)
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
```


File Processing

- *Opening a file* => associating a file on disk with an object in memory.
 - `<filevar> = open(<name>, <mode>)`
 - Associate a disk file <name> with a file object <filevar>
 - <mode> is either 'r' or 'w'
`infile = open("numbers.dat", "r")`
 - We can manipulate the file by manipulating this object.
- *Closing the file* causes any outstanding operations and other bookkeeping for the file to be completed.
 - In some cases, not properly closing a file could result in data loss.
`infile.close()`

Looping through Files

- readline can be used to read one line at a time

```
infile = open(someFile, "r")
for i in range(5):
    line = infile.readline()
    print line[:-1]
```

reads the first 5 lines of a file

Slicing used to strip out the newline char at end of lines

Writing File

- `outfile = open("mydata.out", "w")`
 - Opening a file for writing prepares the file to receive data
- If file already exists
 - file's contents are erased (& starts with empty file).
- If file does not exist
 - new file is created
- `print(<expressions>, file=outfile)`

Modules

- When a Python program starts it only has access to a basic functions and classes.

(“int”, “dict”, “len”, “sum”, “range”, ...)

- “Modules” contain additional functionality.
- Use “import” to tell Python to load a module.

```
>>> import math
```

```
>>> import random
```

“import” vs “from ... import ...”

```
>>> import math
```

```
math.cos
```

```
>>> from math import cos, pi
```

```
cos
```

```
>>> from math import *
```

import the math module

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.cos(0)
1.0
>>> math.cos(math.pi)
-1.0
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
>>> help(math)
>>> help(math.cos)
```

Creating Modules (Library): A Short Introduction

- Modules can be created very easily
 - any script file will be considered a module if it is imported by another script file.
- For example,
 - a script file named `helloworld.py` has the following line
 - `print("Hello World")`
 - another script file in the same directory can import `helloworld.py` by simply referring to its file name.
 - `import helloworld`
- When a script file is imported like a module, all its defined **functions** and **classes** will be available to the importer.
- Any statements in the **top-level** of the script file will be executed as well

Installing modules. Using Package Manager

- PyPI(the Python Package Index)는 Python SW들이 모여있는 저장소
 - 파이썬 개발자들은 자신들의 개발한 파이썬 모듈들을 PyPI에 upload
 - PyPI에 저장된 모듈들은 누구에게나 공개
 - PyPI 홈페이지에 접속하지 않고 pip을 통해서 손쉽게 원하는 모듈을 다운로드
- 먼저 “pip” sw를 pc에 install 해야 한다
- “pip” fetches package meta-data and source codes from an official third-party repository called “PyPI”
- Windows cmd창 or Linux shell 에서:
 - `pip install <package name>`
 - Now you can use the `<package name>` library using `import`

Installing Modules: The Manual Way

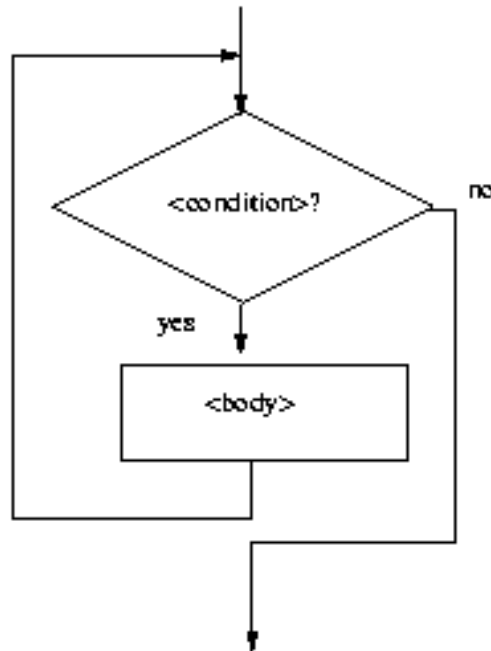
- If you happen to have downloaded Python modules from the internet, or you simply want to use one that you made yourself,
 - Copy the module file into `C/Python34/Lib/site-packages/`
- Note that the file name is case-sensitive, (although it is not in Windows file explorer)
- For example, Zelle's `graphics.py` may be installed in Python
 - `http://mcsp.wartburg.edu/zelle/python/graphics.py`
- Now you can use the graphics library using `import`
 - `import graphics`

Loop Structures

- Loop types classified by **breaking condition** (kind of...) (by Zelle)
 - **Interactive Loop**: user dictates whether to continue the loop interactively
 - **Sentinel Loop**: loop is carried on until certain condition is met
 - **End-of-file Loop**: looped until end of the file.
 - **Nested Loop**: loop in a loop
- Loop structures
 - For loop (definite loops)
 - While loop (indefinite loops)

Indefinite Loops

```
while <condition>:  
    <loop body>
```



- a while loop that counts from 0 to 10:

```
i = 0  
while i <= 10:  
    print(i)  
    i = i + 1
```

- for loop that has the same output

```
for i in range(11):  
    print(i)
```

'WHILE' LOOP

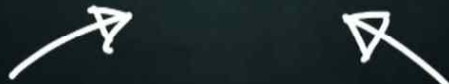
while (condition is true):
do this over and over

```
x = 0
while (x < 10):
    x += 1
```

'BREAK' LOOP

- Used to STOP loop

```
while (true):
    if (something):
        break
```



```
>>> x, y = 0, 0
\
>>> while (True):
    x += 1
    y += 2
    if (x + y > 10):
        break
```

```
>>> x
4
>>> y
8
```

```

Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC v
4 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more in
on.
>>> mail = 0
>>> if mail:
    print 'mail time'
else:
    print 'no mail :(

no mail :(
>>>

```

```

>>> if (7 <= 6):
    print "whaaaaa"
else:
    print "7 is GREATER than 6"

7 is GREATER than 6
>>> if (7) and (6):
    print 'yep'

yep
>>> if (0) and (4):
    print 'wahaaa'

>>> if not (0):
    print 'yep'

yep
>>>

```

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC v
4 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more in
on.
>>> x = [1,2,7]
>>> for i in x:
    print i

1
2
7
>>> |
```

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> for i in range(30):
    print i

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
>>> for i in range(10,30,2):  
    print i
```

```
10  
12  
14  
16  
18  
20  
22  
24  
26  
28
```

```
>>>
```

```
Python Shell  
File Edit Shell Debug Options Windows Help  
>>> for i in range(30):  
    if not (i % 3):  
        continue  
    print i
```

```
1  
2  
4  
5  
7  
8  
10  
11  
13  
14  
16  
17  
19  
20  
22
```

Indefinite Loops: Warning

- The while statement is simple and powerful, yet dangerous! (\exists Zelle)

```
i = 0
while i <= 10:
    print(i)
```

- It can be easy to omit the incrementing logic (applies to all languages)

Interactive Loops: Average computation example

- Basic pseudocode

```
set moredata to "yes"
while moredata is "yes"
    get the next data item
    process the item
    ask user if there is moredata
```

- Example

```
def main():
    moredata = "yes"
    sum = 0.0
    count = 0
    while moredata[0] == 'y':
        x = eval(input("Enter a number >> "))
        sum = sum + x
        count = count + 1
        moredata = input("More numbers (yes or no)? ")
    print("\nThe average of the numbers is", sum / count)
```


Interactive Loops: Example

Enter a number >> 32

Do you have more numbers (yes or no)? y

Enter a number >> 45

Do you have more numbers (yes or no)? yes

Enter a number >> 34

Do you have more numbers (yes or no)? yup

Enter a number >> 76

Do you have more numbers (yes or no)? y

Enter a number >> 45

Do you have more numbers (yes or no)? nah

The average of the numbers is 46.4

Sentinel Loops

- Continues to process data until reaching a special value (called the sentinel **보초, 파수병**) that signals the end.
- The sentinel must be distinguishable from the data
 - since it is not processed as part of the data.

```
# average3.py
#     A program to average a set of numbers
#     Illustrates sentinel loop using negative input as sentinel

def main():
    sum = 0.0
    count = 0
    x = eval(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum = sum + x
        count = count + 1
        x = eval(input("Enter a number (negative to quit) >> "))
    print("\nThe average of the numbers is", sum / count)
```

➤ Assuming there are no negative numbers in the data.

Sentinel Loops: Using empty string as the sentinel

```
# average4.py
#     A program to average a set of numbers
#     Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")

    print("\nThe average of the numbers is", sum / count)
```

File Loops: Example

```
# average6.py
#     Computes the average of numbers listed in a file.

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName, 'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        sum = sum + eval(line)
        count = count + 1
        line = infile.readline()

    print("\nThe average of the numbers is", sum / count)
```

Nested File Loops: Example

- We want to read any number of numbers on a line in the file (separated by commas)
- 3, 4, 5, 6, 1, 2, ..., 1
- 3, 2, 1, 7, 5, 2, ..., 1
- 5, 6, 4, 7, 5, 6, ...,
- We use two loops:
 - The top-level loop loops through each line of the file
 - The second-level loop loops through each number of each line

```
# average7.py
#     Computes the average of numbers listed in a file.
#     Works with multiple numbers on a line.

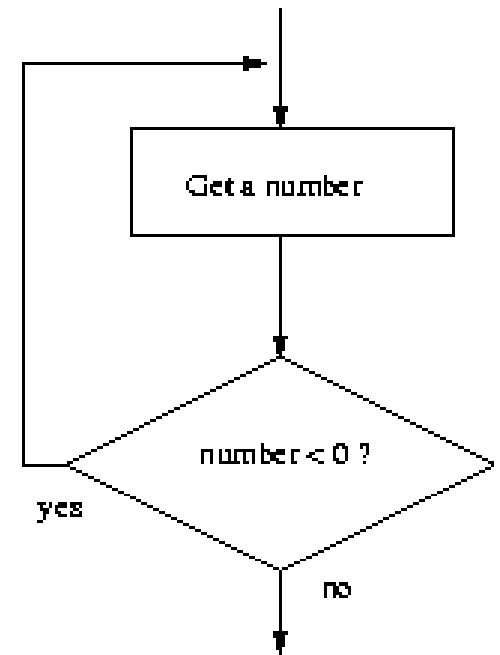
import string
def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName, 'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        for xStr in line.split(","):
            sum = sum + eval(xStr)
            count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", sum / count)
```

Post-Test Loops

- Condition test comes after the body of the loop
 - A post-test loop always executes the body of the code at least once.
- Python doesn't have a built-in statement to do this,
 - but we can do it with a slightly modified **while** loop.
 - In some other languages (not in Python);

repeat

get a number from the user



Post-Test Loops: Example

- Using a **while** statement
 - Seed the loop condition so we're guaranteed to execute the loop once.

```
number = -1
while number < 0:
    number = eval(input("Enter a positive number: "))
```

- By setting number to -1, we force the loop body to execute at least once.
- The same algorithm implemented with a break:

```
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0: break
```

- Executing break causes Python to immediately exit the enclosing loop.

Composite Boolean Expressions

- Boolean operators and, or, not.

<expr> and <expr>

<expr> or <expr>

not <expr>

- We can represent their semantics (meaning) using a *truth table*.

P	Q	$P \text{ and } Q$	$P \text{ or } Q$	$\text{not } Q$
T	T	T	T	F
T	F	F	T	T
F	T	F	T	–
F	F	F	F	–

- The order of operators: **not** > **and** > **or**
- For example,
 - a or not b and c
 - (a or ((not b) and c))

Boolean Algebra

- Anything `ored` with `true` is `true`:

`a or true == true`

- Both `and` and `or` distributive:

`a or (b and c) == (a or b) and (a or c)`

`a and (b or c) == (a and b) or (a and c)`

- Double negatives cancel out:

`not(not a) == a`

- DeMorgan's laws:

`not(a or b) == (not a) and (not b)`

`not(a and b) == (not a) or (not b)`

Boolean Expressions: Evaluating Other Types

- Python will let you evaluate any built-in data type as a Boolean.
- For numbers (int, float, and long ints), zero is considered False, anything else is considered True.
- An empty sequence is interpreted as False while any non-empty sequence is taken to mean True.
- The Boolean operators have operational definitions that make them useful for other purposes.

```
>>> bool(0)
False
>>> bool(32)
True
>>> bool("")
False
>>> bool([])
False
>>> bool(1)
True
>>> bool("Hello")
True
>>> bool([1,2,3])
True
```

Boolean Expressions: As Short-Circuit Operators

- Boolean operators can have operational definitions that make them useful for other purposes

Operator	Operational definition
x and y	If x is false, return x . Otherwise, return y .
x or y	If x is true, return x . Otherwise, return y .
not x	If x is false, return True. Otherwise, return False.

- Python's Booleans are *short-circuit operators*
 - meaning that a true or false is returned as soon as the result is known.
- Python will not evaluate the second expression;
 - in an **and** where the first expression is false and
 - in an **or**, where the first expression is true
- Internally, Python evaluates Boolean operators with any built-in types of operands, as any built-in types can be cast to *bool*.

Boolean Expressions

- Suppose that a student intends to check if user's input is a yes. He wrote

`response == "y" or "Y"`

- What is the evaluation result of the expressions if `response = "y"`? What if `response = "Y"`?
- Above expression will be evaluated as `(response == "y") or "Y"`.
 - if `response == "y"` is True, True is returned.
 - if `response == "y"` is not True, "Y" is returned.
- A simpler way is

`response.lower() == "y"`

Little bit Advanced Features

SIMPLE FUNCTION

```
def createFile(dest):  
    print dest  
  
if __name__ == '__main__':  
    createFile('ham')  
    raw_input('done!!')
```

Will be string

is this file the main file?

SAVE!!!

createTextFile.py

createTextFile.py를 Python interpreter에서 수행하면

(즉 `python createTextFile.py` 하면)

`if __name__ == '__main__':` 이 **true**가 되고 그 아래 문장들이 수행됨

반면에 `import createTextFile` 하면

`if __name__ == '__main__':` 이 **false**가 되고 그 아래 문장들이 수행이 안됨

```
74 "createTextFile.py - C:/Users/The_Captain/Desktop/LetsLearn/createTextFile.py"
File Edit Format Run Options Windows Help

import time as t
from os import path

def createFile(dest):
    """
    The script creates a text file at the passed in location
    names file based on date
    """
    date = t.localtime(t.time())

    ## FileName = Month_Day_Year
    name = '%d_%d_%d.txt' % (date[1], date[2], (date[0] % 100))

    if not (path.isfile(dest + name)):
        f = open(dest + name, 'w')
        f.write("\n" * 30)
        f.close()

if __name__ == '__main__':
    destination = 'C:\\Users\\The_Captain\\Desktop\\LetsLearn\\'
    createFile(destination)
    raw_input("done!!!")
```

CREATING A TEXT FILE

Goal:

- Create a "Month_Day_Year.txt" file
- Within file, 30 blank lines

```
...
date = t.localtime(t.time())
name = "%d_%d_%d.txt" % (date[1],
                           date[2], (date[0] % 100))
...
```

get current time/date

output in 'Month_Day_Year.txt' format

SIMPLE FUNCTION

```
def createFile(dest):  
    print dest  
  
if __name__ == '__main__':  
    createFile('ham')  
    raw_input('done!!')
```

Will be string

is this file the main file?

SAVE!!!

createTextFile.py

createTextFile.py를

Python interpreter에서 수행하면

(즉 window의 cmd창에서 `>> python createTextFile.py` 하면)

If `__name__ == '__main__':` 이 `true`가 되고 그 아래 문장들이 수행됨

다른 python program에서 `import createTextFile` 하면

if `__name__ == '__main__':` 이 `false`가 되고 그 아래 문장들이 수행이 안됨

** `__name__` 은 python의 special variable로써 나를 부른 program의 이름을 가지고 있음

DICTIONARY EXAMPLE

```
sam = {}  
sam["weapon"] = "chainsaw"  
sam["health"] = 10
```

DICTIONARY EXAMPLE

dictionary[key]: GET and SET the value
del dict[key]: DELETE a value/key pair

```
sam["weapon"]  
del sam["health"]
```

Python 2.7.3 (default, Apr 10 2012, 23:31:26) [M
32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for mor
on.

```
>>> sam = {}  
>>> sam["weapon"] = "chainsaw"  
>>> sam["health"] = 10  
>>> sam  
{'weapon': 'chainsaw', 'health': 10}  
>>> sam["weapon"]  
'chainsaw'  
>>> del sam["health"]  
>>> sam  
{'weapon': 'chainsaw'}  
>>>
```

```
myDict ➔ {key1: value1, key2: value2, key3: data3, ...}  
myDict[key8] = value13    # add a "key8:value13" pair  
myDict[key2]              # retrieve the value part of  
key2  
del myDict[key5]          # delete the "key5:data5" pair
```

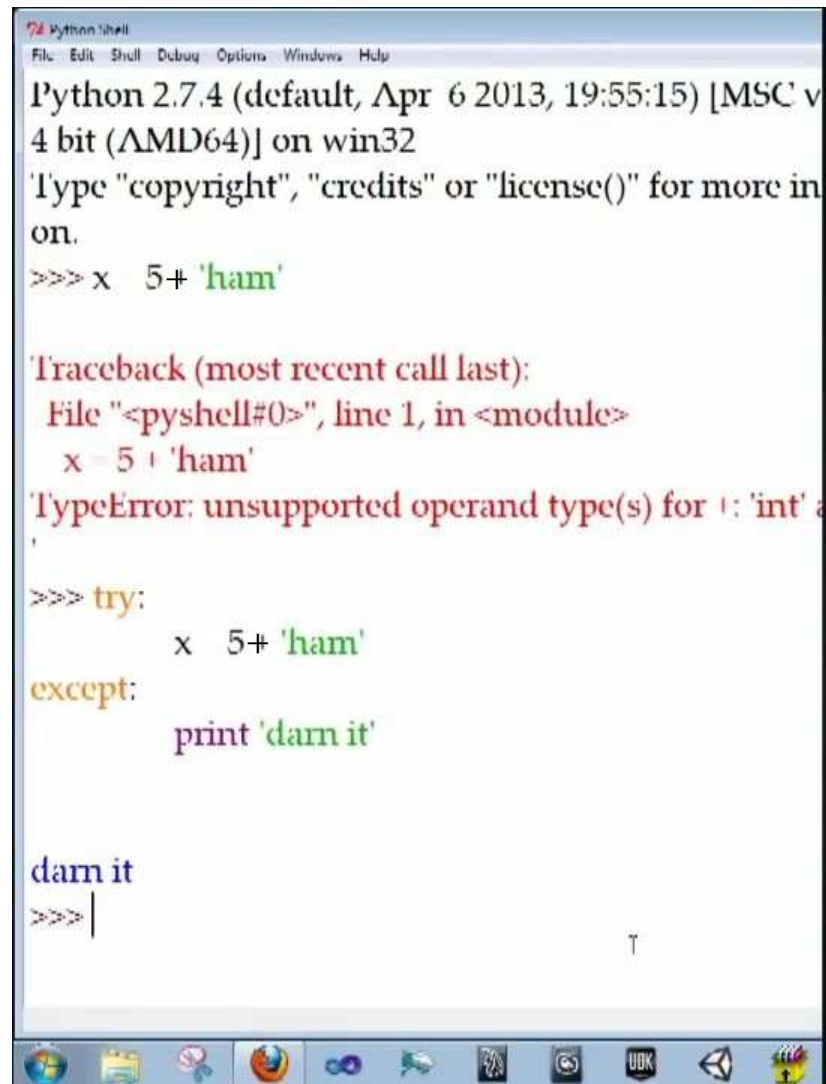
Exception Handling

TRY

- 'try' TO EXECUTE the code below...
- May be used anywhere that keyboarduser input is required

EXCEPT

- CATCHES all ERRORS or can just catch a specific error
- May be used anywhere that keyboarduser input is required



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC v
4 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more in
on.
>>> x = 5 + 'ham'

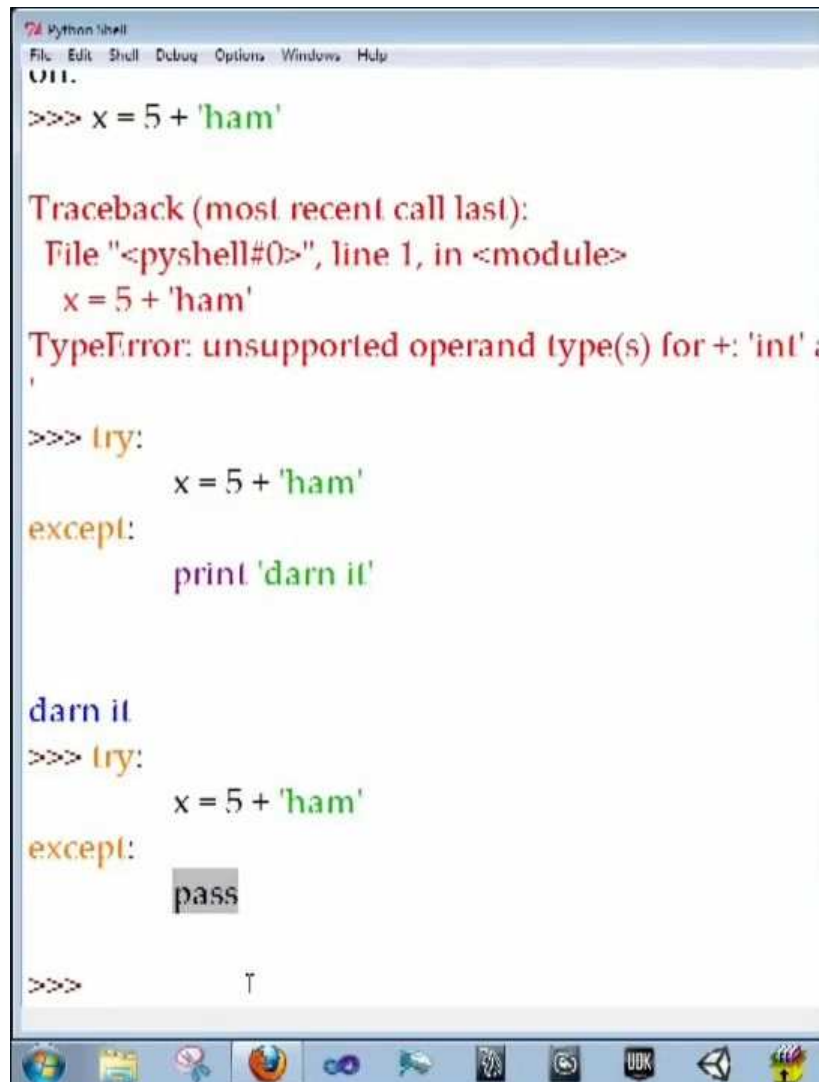
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    x = 5 + 'ham'
TypeError: unsupported operand type(s) for +: 'int' a
,

>>> try:
        x = 5 + 'ham'
except:
        print 'dam it'

dam it
>>> |
```

PASS

- says to IGNORE and move on
- may be used in For, While, Try/Except instances



```
Python Shell
File Edit Shell Debug Options Windows Help
OFF.
>>> x = 5 + 'ham'

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    x = 5 + 'ham'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> try:
      x = 5 + 'ham'
except:
      print 'darn it'

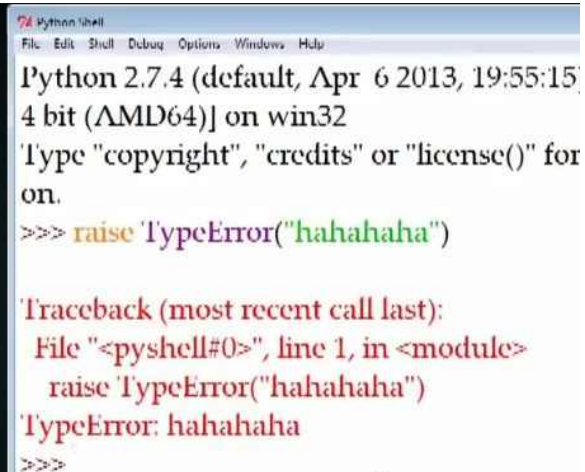
darn it
>>> try:
      x = 5 + 'ham'
except:
      pass

>>>
```

RAISE

- FORCE AN ERROR
to occur

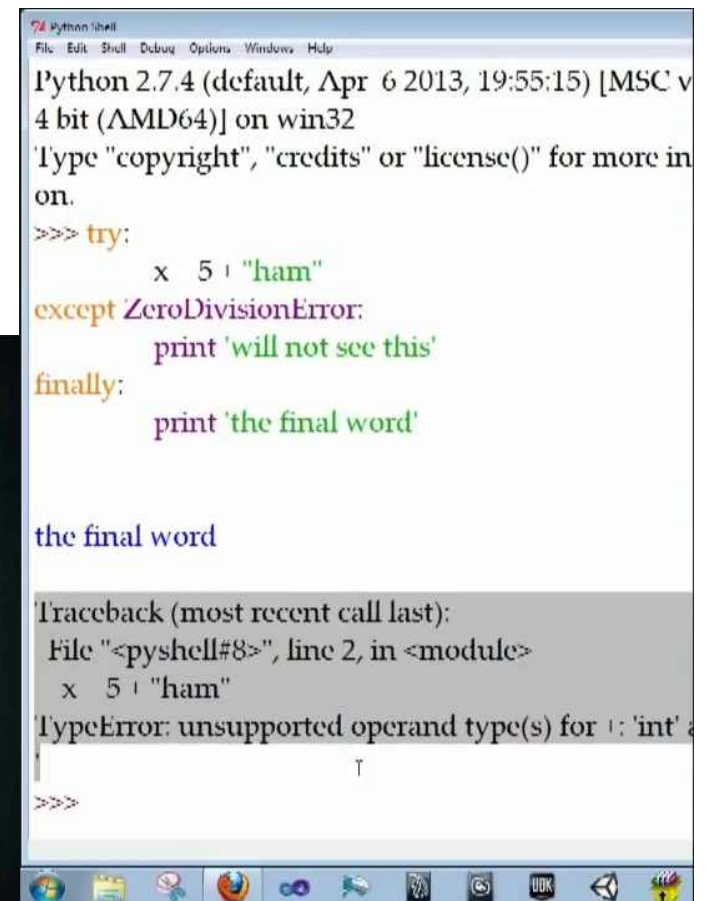
```
raise TypeError("hahaha")
```



```
Python 2.7.4 (default, Apr 6 2013, 19:55:15)
4 bit (AMD64) on win32
Type "copyright", "credits" or "license()" for
on.
>>> raise TypeError("hahahaha")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise TypeError("hahahaha")
TypeError: hahahaha
>>>
```

FINALLY

```
try:
    x = 5 + 'ham'
except ZeroDivision:
    print 'will not see this'
finally:
    print 'the final word'
```



```
Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC v
4 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more in
on.
>>> try:
        x = 5 + "ham"
    except ZeroDivisionError:
        print 'will not see this'
    finally:
        print 'the final word'

the final word

Traceback (most recent call last):
  File "<pyshell#8>", line 2, in <module>
    x = 5 + "ham"
TypeError: unsupported operand type(s) for +: 'int' a
>>>
```

Exception Handling

- A mechanism to handle exceptional problems.
 - It eliminates the need to check at each step of the algorithm

```
try:
    <body>
except:
    <exception handling>
```

- To explicitly filter out all error types

```
try:
    <body>
except <error_1> :
    <exception handling>
...
except <error_n> :
    <exception handling>
```

- To explicitly filter out error types and store the error as a variable

```
try:
    <body>
except <error_1> as <variable_1> :
    <exception handling>
...
except <error_n> as <variable_n> :
    <exception handling>
```

Exception Handling: Example

```
# quadratic5.py
#     A program that computes the real roots of a quadratic equation.
#     Illustrates exception handling to avoid crash on bad inputs

import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    try:
        a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\n The solutions are:", root1, root2)
    except ValueError:
        print("\n No real_number roots")
```

Exception Handling

- Full list of standard built-in exceptions (users may create their own) is listed here.

<https://docs.python.org/3/library/exceptions.html>

- In the quadratic equation example, other types of exceptions may arise
 - not entering the right number of parameters (“unpack tuple of wrong size”),
 - entering an identifier instead of a number (`NameError`),
 - entering an invalid Python expression (`TypeError`).

ARGUMENT TYPES

Regular
Argument

Keyword
Argument

`def myFunc(var1, var 2 = 3):`

...

Keyword args set
DEFAULT value that
MAY be overridden

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC v
64)] on win32
Type "copyright", "credits" or "license()" for more in
>>> def doesNothing():
    pass

>>> doesNothing()
>>> def makeOne():
    return 1

>>> x = makeOne()
>>> print x
1
>>> |
```


LOCAL VS GLOBAL VARIABLES

GLOBAL: variable that accessible
ANYWHERE within program.

Uses keyword 'global'

glVar = 5

def myFunc():

global glVar

```
>>> def makeOne():  
    return 1  
  
>>> x = makeOne()  
>>> print x  
1  
>>> def addTen(myInt):  
    myInt += 10  
    return myInt  
  
>>> x = 12  
>>> dir()  
['__builtins__', '__doc__', '__name__', '__package__',  
'othing', 'makeOne', 'x']  
>>> y = addTen(x)  
>>> print x,y  
12 22  
>>>
```

DOCUMENT STRING

- Text DESCRIBING the function
- Comes immediately after function creation
- Use triple quotes to enclose

```
def myFunc():
```

```
    """
```

```
    My description
```

```
    """
```

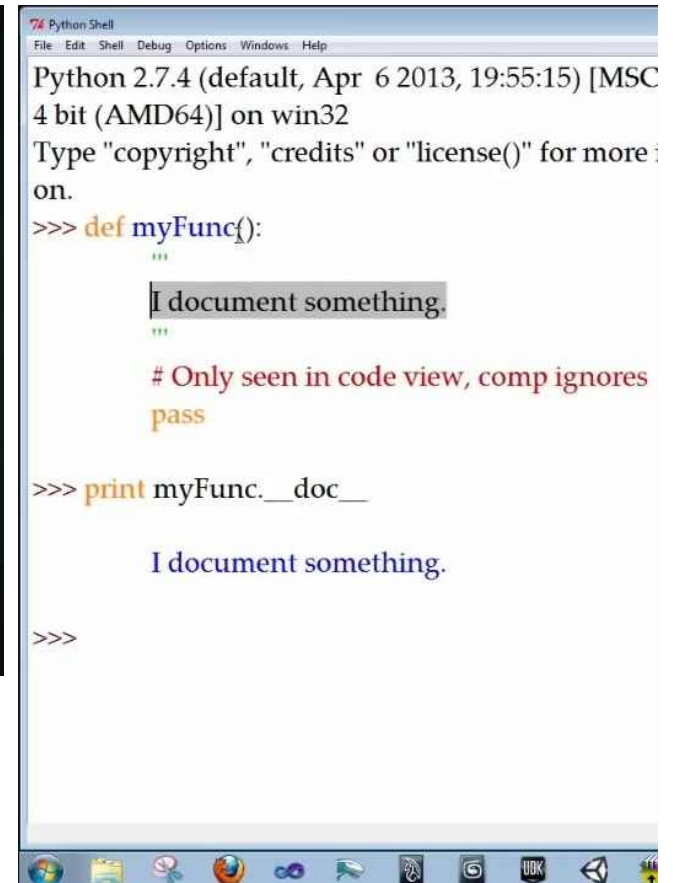
COMMENTS

- Tell program to IGNORE everything afterward in line
- declared with '#' pound/sharp symbol
- Frequently used to write notes or 'ignore' bits of code

```
# comment 1
```

```
x = 5 #2
```

```
#3
```



```
Python Shell
File Edit Shell Debug Options Windows Help

Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC
4 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more
>>> def myFunc():
    """
    I document something.
    """
    # Only seen in code view, comp ignores
    pass
>>> print myFunc.__doc__

I document something.
>>>
```