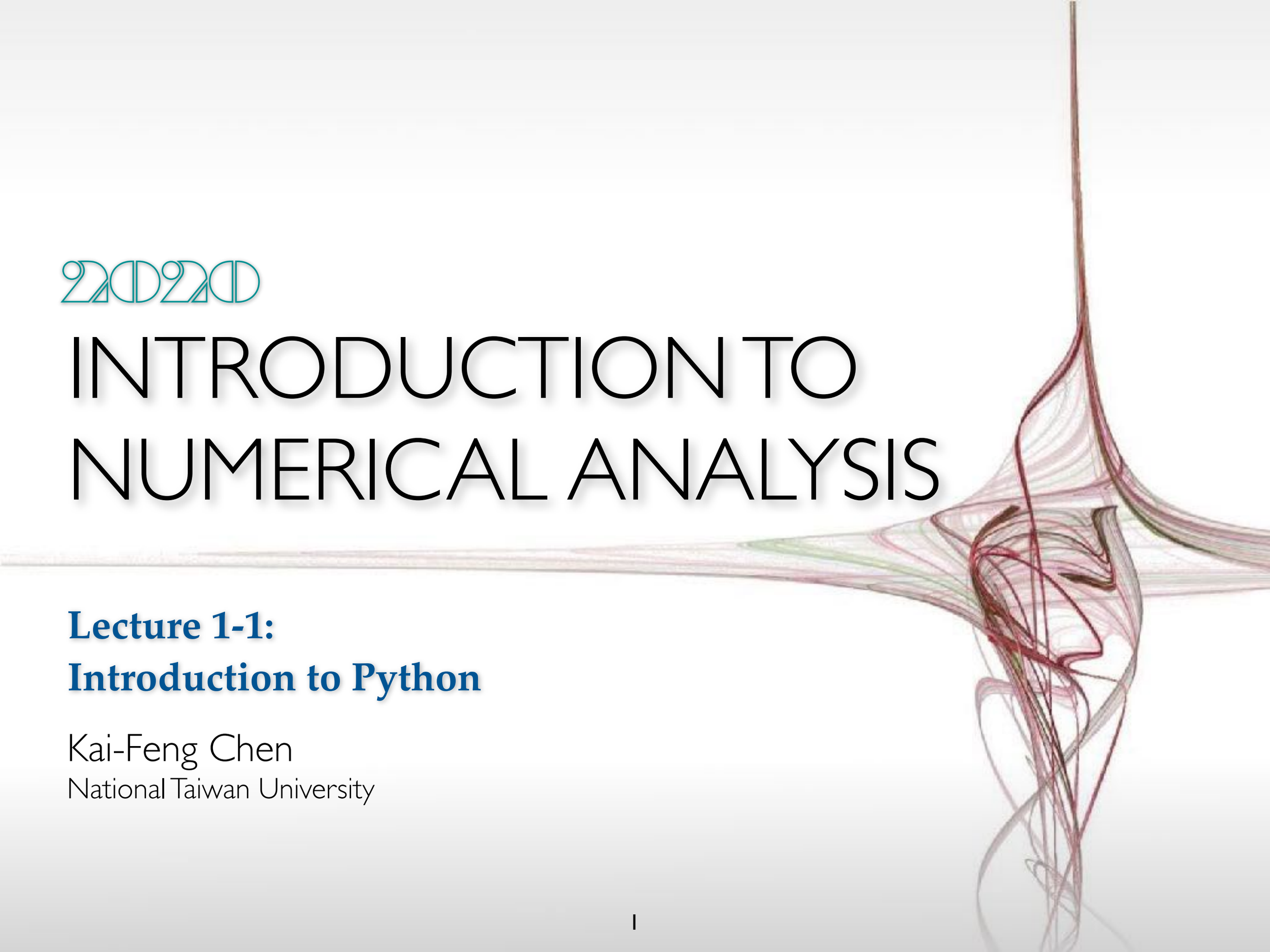
2020

INTRODUCTION TO NUMERICAL ANALYSIS



Lecture 1-1: Introduction to Python

Kai-Feng Chen
National Taiwan University

PYTHON: AN INTRODUCTION

- Quote from Wikipedia: "Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. The language provides constructs intended to enable clear programs on both a small and large scale."
- The core philosophy (as called "The Zen of Python"):
  - Beautiful is better than ugly.
  - Explicit is better than implicit.
  - Simple is better than complex.
  - Complex is better than complicated.
  - Readability counts.

In short:
Must be simple and beautiful

PYTHON: AN INTRODUCTION (II)

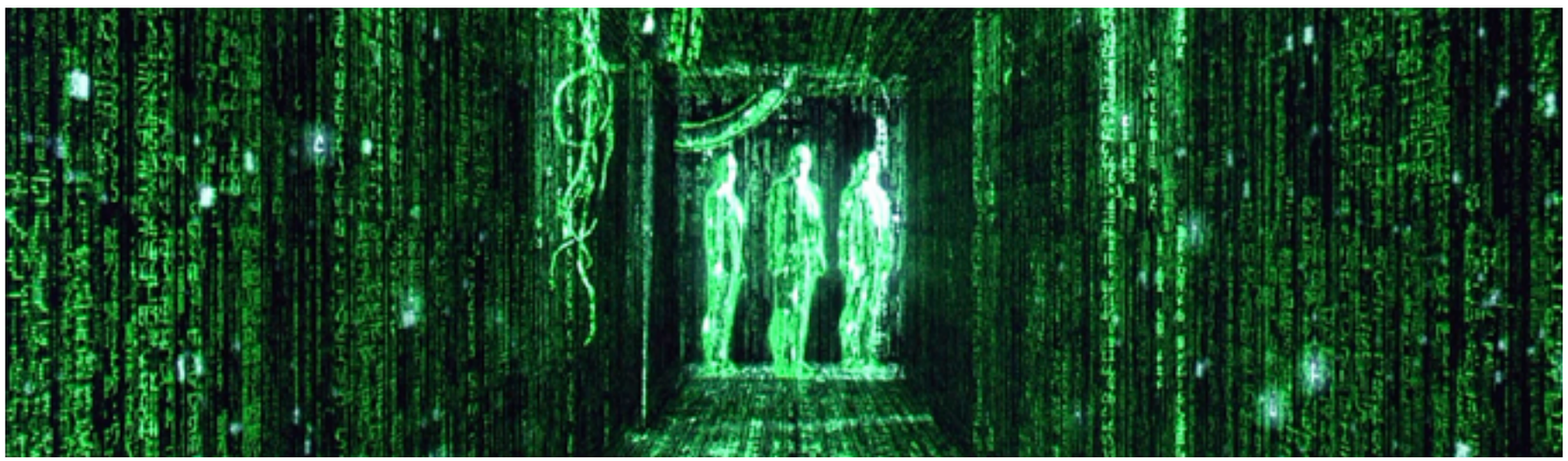
- Python development was started in December 1989.
- Python's principal author: Guido van Rossum.
- The origin of the name comes from Monty Python (Monty Python's Flying Circus, a British television comedy show), ***not the elongated legless animal***.

Guido van Rossum

Monty Python's Flying Circus

PYTHON,
THE PROGRAMMING LANGUAGE

- Python is a **high-level language**; others: C/C++, Fortran, Java, etc.
- The computers can only run the "machine language", which is not human readable (well, if you are a real human).
- The programs written in a high-level language are shorter and easier to read, and they are portable (not restricted to only one kind of computer). However, they have to be processed ("translated") before they can run.



"HELLO WORLD" IN C/C++

Well, the common starting point for any programming language.

```c
#include <stdio.h>

int main()
{
  printf("hello world!\n");
}
```

helloworld.cc

- If you run it in a unix-like system with a **gcc compiler**:

```
% gcc helloworld.cc
% ./a.out
hello world!
%
```

"HELLO WORLD" IN PYTHON

```
print('hello world!')
```

helloworld.py

- Run it in your terminal with **python interpreter**:

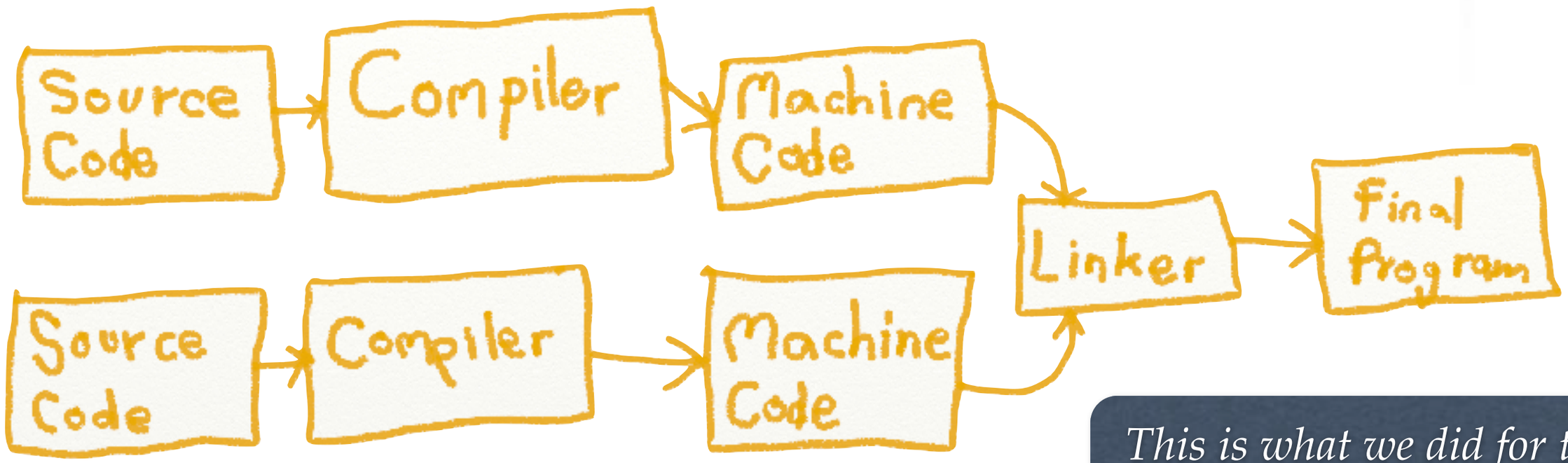
```
% python helloworld.py
hello world!
%
```

*All you need is just **a single line**!*

*The standard filename extension for python is "**.py**".*

THE COMPILER (& LINKER)

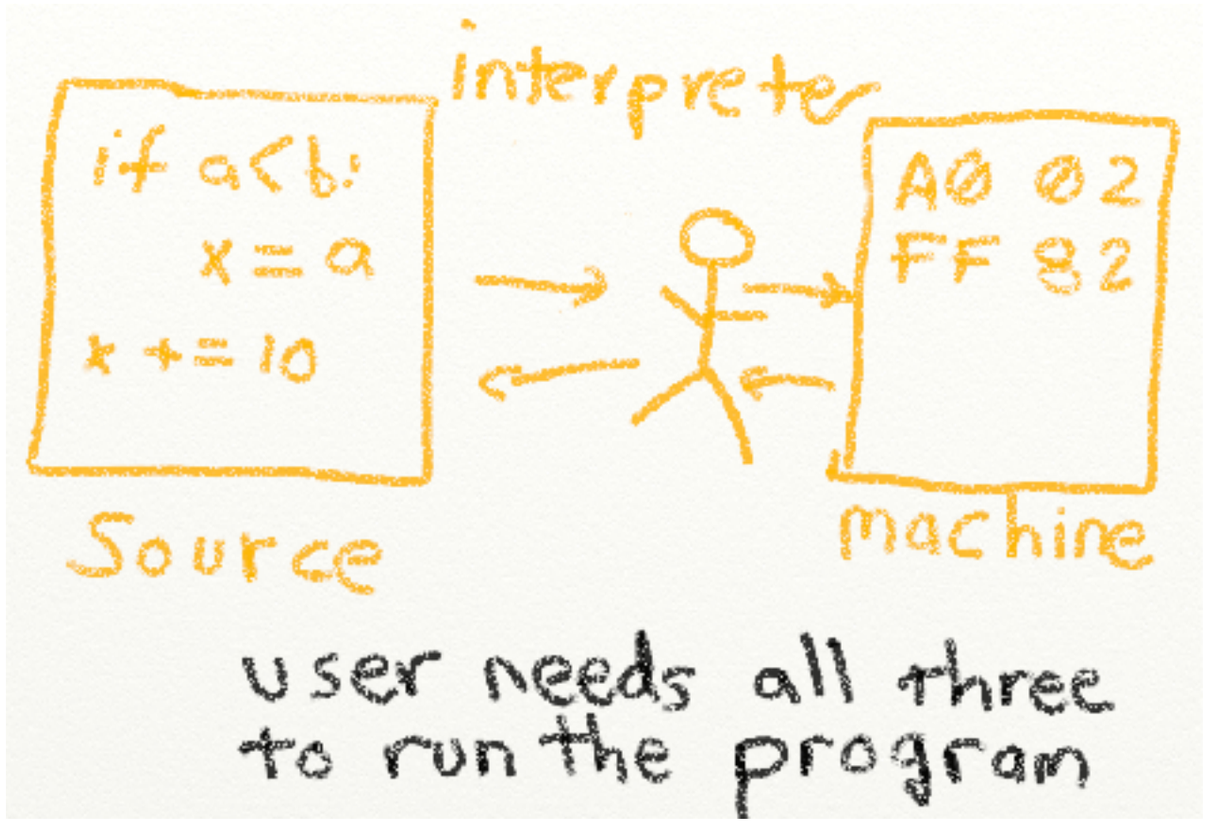
- A compiler reads the **source code** and translates it before the program starts running; the translated program is the **object code** or the **executable**.
- Once a program is compiled, it can be executed it repeatedly without further translation.



This is what we did for the C version of "hello world".

THE INTERPRETER

- An interpreter reads a high-level program (the source code) and executes it.
- It processes the program little-by-little at a time, reading lines and performing computations.



The python version of "hello world" program is executed by the python interpreter.

PYTHON, THE INTERPRETER

- Python is considered an **interpreted language** because Python programs are executed by an interpreter.
  - Rapid turn around: no needs of compilation and linking as in C or C++ for any modification.
  - Hybrid Python programs are compiled automatically to an intermediate form called **bytecode**, which the interpreter then reads.
  - This gives Python the development speed of an interpreter without the performance loss in purely interpreted languages.
  - However for serious speed limited computations you still need to use C, or even Fortran.

PYTHON, THE INTERPRETER (II)

- There are actually two ways to use the interpreter: **interactive mode** and **script mode**.
- Running your program in script mode:
  - As we did in the previous "hello world" example: put your code into a file, process it with the python interpreter.
  - The Unix/Linux "#!" trick:

```
#!/usr/bin/env python
print('hello world!')
```

helloworld.py

```
% chmod +x helloworld.py   ⇐ Make it as an executable file
% ./helloworld.py
hello world!
```

PLAY WITH PYTHON INTERACTIVELY (I)

- Let's try some simple Python commands in the interactive mode:

```
% python
>>> 2+2
4
>>> # I'm a comment. I will not do anything here.
... 2+2 # anything after '#' is a comment.
4
>>> 6/3
2.0
>>> (50-5*6)/4
5.0
>>> 6/-3
-2.0
>>> 2**8  ⇐ this is exponentiation (2⁸)
256
>>> 10%3  ⇐ this is modulus
1
```

Use the python interpreter as a calculator!
The regular precedence is followed:
(,) ⇒ ** ⇒ *,/,// ⇒ % ⇒ +,-

PLAY WITH PYTHON INTERACTIVELY (II)

- Float point number versus integer:

```
>>> 8/3
2.6666666666666665 ⇐ automatically convert to float since python 3
>>> 8//3 ⇐ integer divide
2
>>> 9998//9999 ⇐ round to integer
0
>>> 8/3.
2.6666666666666665
>>> 8/(6.6/2.2)
2.6666666666666667
>>> 8/3+1
3.6666666666666665
>>> round(3.1415927,3)  ⇐ rounding off to 3rd digit
3.142
```

PLAY WITH PYTHON INTERACTIVELY (III)

- You can assign a **variable** with "=":

```
>>> width = 20
>>> height = 5*9
>>> area = width * height
>>> area
900
```

- You can assign multiple variables as well:

```
>>> x = y = z = 1
>>> x
1
>>> x + y + z
3
```

PLAY WITH PYTHON INTERACTIVELY (IV)

- Complex number is also supported. The imaginary part can be written with a suffix "j".

```
>>> a = 2j
>>> a*a
(-4+0j)
>>> b = complex(3,4) ⇐ Using the function complex(real, imag)
>>> a+b
(3+6j)
>>> 8j/3j
(2.6666666666666665+0j) ⇐ Complex numbers are always float.
>>> b.real + b.imag ⇐ Exact the real/imaginary part
7.0                    with .real and .imag
>>> abs(b) ⇐ absolute value
5.0
```

FIRST VISIT WITH STRINGS

- Python supports for strings are actually very powerful. See several examples below:

```
>>> 'I am a string!'
'I am a string!'
>>> "me too!"
'me too!'
>>> 'anything quoted with two \' is a string.'
"anything quoted with two ' is a string."
>>> "\" also works."
'" also works.'
>>> "pyt" 'hon'
'python'
>>> sentence = 'I am a string variable.'
>>> sentence
'I am a string variable.'
```

⇐ Two string literals next to each other are automatically concatenated

FIRST VISIT WITH STRINGS (II)

- More: multiline strings

```
>>> lines = "line1\nline2\nline3"
>>> print lines
line1
line2
line3
>>> reply = '''I    ⇐ Use ''' or """ to start/end multiline strings.
... also have
... 3 lines'''
>>> print reply
I
also have
3 lines
```

FIRST VISIT WITH STRINGS (III)

- More: operators "+", "*", and type conversions

```
>>> 'oh'+'my'+'god'
'ohmygod'
>>> 'It\'s s'+'o'*20+' delicious!!'
"It's soooooooooooooooooooo delicious!!"
>>> '1234'+'1234'
'12341234'
>>> int('1234')+int('1234')
2468
>>> float('1234')/100
12.34
>>> a = b = 2
>>> str(a)+' plus '+str(b)+' is '+str(a+b)
'2 plus 2 is 4'
```

We will talk more on python strings afterwards.

VALUES AND TYPES

- A **value** is one of the basic things a program works with, like a **letter** or a **number**.
  - 1234 is an integer. 1234 ⇒ value; integer ⇒ type.
  - ‘hello’ is a string: ‘hello’ ⇒ value; string ⇒ type.
- The interpreter can also tell you what type a value has:

```
>>> type('hello!')
<class 'str'>
>>> type(1234)
<class 'int'>
>>> type(3.14159)
<class 'float'>
>>> type(3+4j)
<class 'complex'>
```

TYPES & MORE

- Type casting is easy:

```
>>> float(1234)
1234.0
>>> type(float(1234))
<class 'float'>
>>> type('1234')
<class 'str'>
>>> type(int('1234'))
<class 'int'>
>>> type(str(1234))
<class ‘str'>
>>> type(complex('3+4j'))
<class 'complex'>
```

TYPES & MORE (II)

- Few more complicated (but very useful) types are also supported. Will be discussed in a near future lecture.

```
>>> type(3<4)
<class 'bool'>
>>> type(open('tmpfile','w'))
<class '_io.TextIOWrapper'>
>>> type([1,2,3,4,['five',6.0]])
<class 'list'>
>>> type((1,2,'three',4))
<class 'tuple'>
>>> type({'name':'Chen','phone':33665153})
<class 'dict'>
>>> type({'a','b','c'})
<class 'set'>
```

VARIABLES

- A **variable** is a name that refers to a value.
- An **assignment statement** creates new variables and gives them values (you have seen this already):

```
>>> pi = 3.1415926536
>>> last_message = 'what a beautiful day'
```

- Programmers generally choose names for their variables that are meaningful – they document what the variable is used for.
- Variable names can be arbitrarily long, can be with both letters and numbers.
- The underscore character "_" can appear in a name as well.

VARIABLES (CONT.)

- The variable names have to start with a letter or "_"
- It is a good idea to begin variable names with **a lowercase letter**.
- **Python's keywords**: keywords are used to recognize the structure of the program, and cannot be used as variable names.

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

INTERMISSION

- Now it's the time to play with your python interpreter:
  - As a simple calculator.
  - Play with the strings.
  - Examine the python built-in types.
  - Define your variables and check any possible break-down cases.

A STEP TOWARD PROGRAMMING

- A **program** is a sequence of instructions that specifies how to perform a mathematical/symbolic computation.
- This is a simple example of mathematical computation:

```
s = t = 1.
for n in range(1,21):
    t /= n  ⇐ t /= n is equivalent to t = t/n
    s += t
print ("1/0!+1/1!+1/2!+1/3!+...+1/20! =",s)
```

```
1/0!+1/1!+1/2!+1/3!+...+1/20! = 2.71828182846
```

THE PRINT STATEMENT

- As you already seen in the “hello world!” program, as well as the previous example, the **print** statement simply present the output to the screen (your terminal).
- A couple of examples:

```
>>> print('show it now!')
show it now!
>>> print(1,2,(10+5j),-0.999,'STRING')  ⇐ connect w/ comma
1 2 (10+5j) -0.999 STRING
>>> print(1,2,3,4,5)
1 2 3 4 5
>>> print(str(1)+str(2)+str(3)+str(4)+str(5))
12345
```

Remark: for python 2, the ‘print’ was not a function, you can use something like print ‘hello world!’.

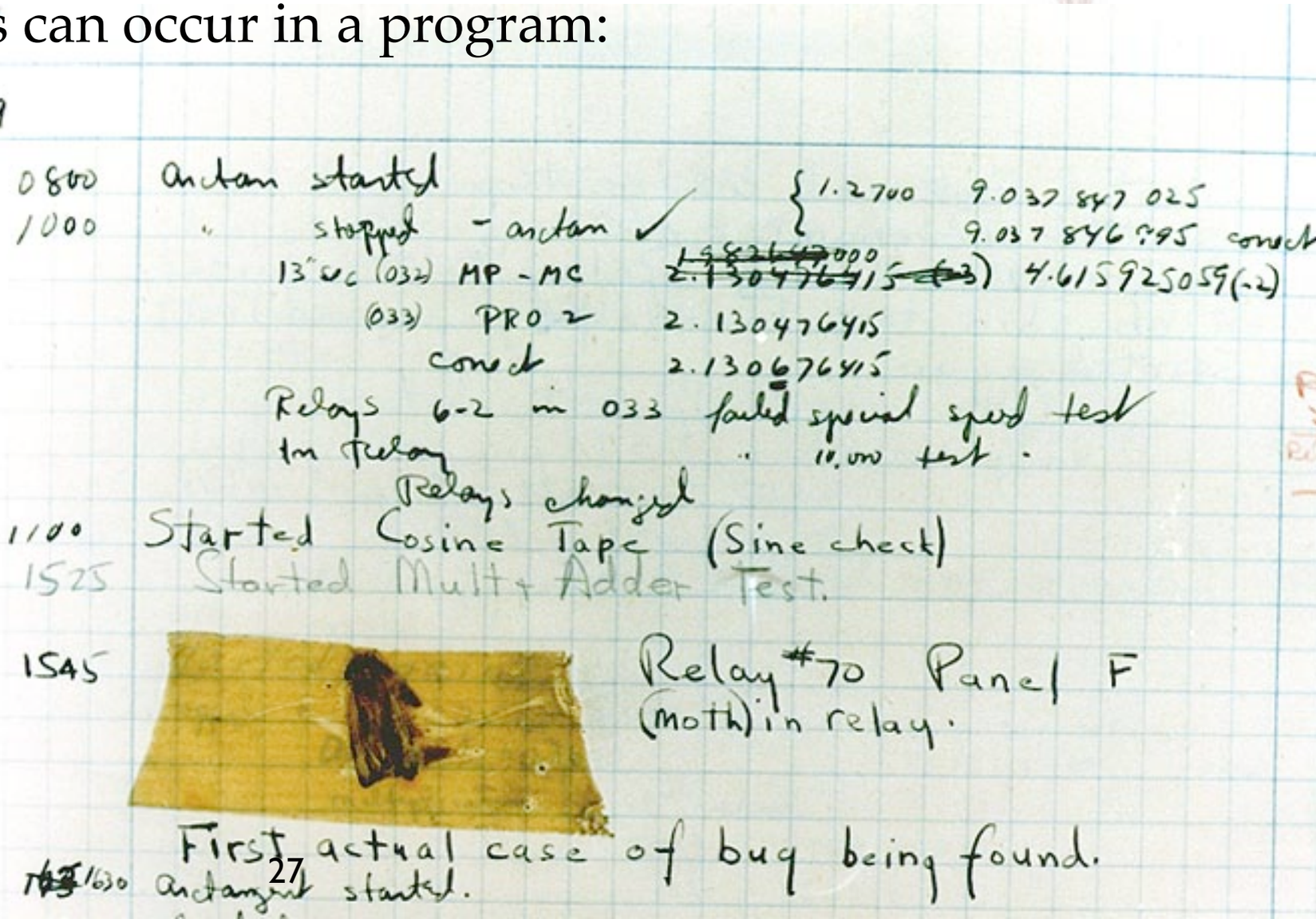
STRUCTURE OF A PROGRAM

- **Input:**
  get data from the keyboard, a file, or some other device.
- **Output:**
  display data on the screen or send data to a file or other device.
- **Math:**
  perform basic mathematical operations like addition and multiplication.
- **Conditional execution:**
  check for certain conditions and execute the appropriate code.
- **Repetition:**
  perform some action repeatedly, usually with some variation.

Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these.

GIVE ME A FLYSWATTER

- Programming is error-prone. Programming errors are called **bugs** and the process of tracking them down is called **debugging**.
- Three kinds of errors can occur in a program:
  - **Syntax errors**
  - **Runtime errors**
  - **Semantic errors**



SYNTAX ERRORS

- Syntax refers to the structure of a program and the rules about that structure.
- Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message.
- An example:

```
>>> y = 1 + 4x
  File "<stdin>", line 1
    y = 1 + 4x
            ^
SyntaxError: invalid syntax
>>>
```

You probably already see a couple of syntax errors in your previous trials.

RUNTIME ERRORS

- A runtime error does not appear until after the program has started running.
- These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.
- The following "ZeroDivisionError" only occur when it runs.

```
>>> a = b = 2
>>> r = ((a + b)/(a - b))**0.5 * (a - b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

SEMANTIC ERRORS

- If there is a semantic error in your program, it will run successfully, but it will not do the right thing.
- It does something else.
- **The problem is that the program you wrote is not the exactly program you wanted to write.**
- The meaning of the program (its semantics) is wrong.
- Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.
- **(Much) more difficult to debug.**

SEMANTIC ERRORS

- As an example:

Just one line with missing intent gives you a completely different answer!

```
□□□□s += t
```

```
s = t = 1.
for n in range(1,21):
    t /= n
s += t
print ("1/0!+1/1!+1/2!+1/3!+...+1/20! =",s)
```

```
1/0!+1/1!+1/2!+1/3!+...+1/20! = 1.0
```

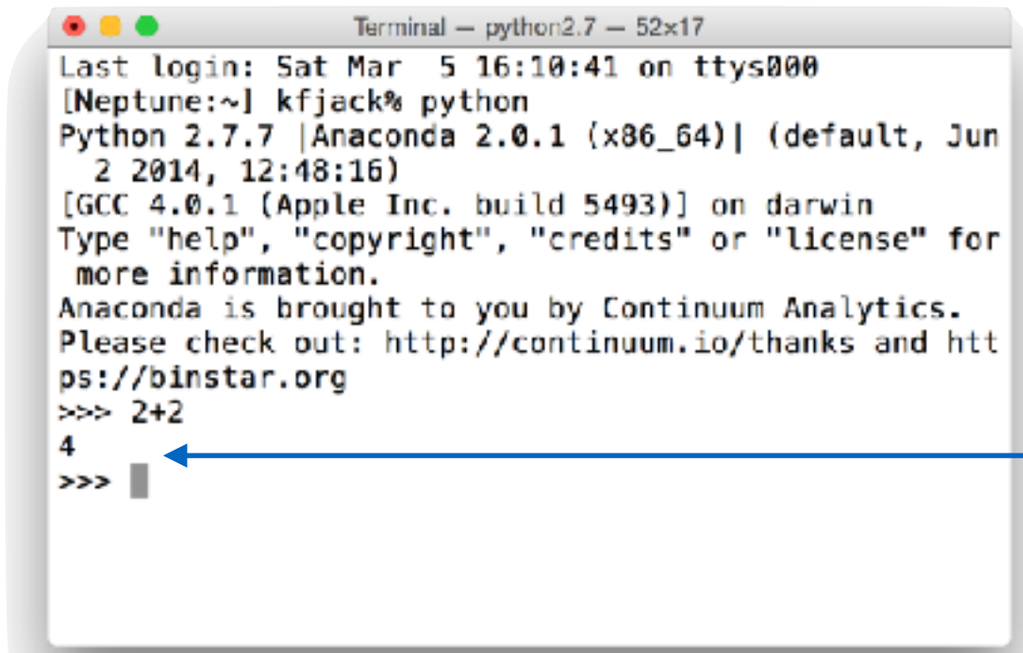
```
2.71828182846
```

DEBUGGING

- It is very difficult to write a bug free program at the first shot.
- Programming and debugging can be the same thing.
  Programming is the process of debugging a program until it does exactly what you want.
- Having a good programming/coding habit sufficiently helps.
- Re-run a buggy program never helps!

SOME COMMON ISSUES

- As observed from previous lectures, there were some common issues from the students…
- **Issue #1: How to use the interpreter mode** — starting the **interpreter mode** you can just type "**python**" in your terminal or in your command line window:



In the slides of this lecture, this is always shown in a blue box.

```
% python
>>> 2+2
4
>>>
```

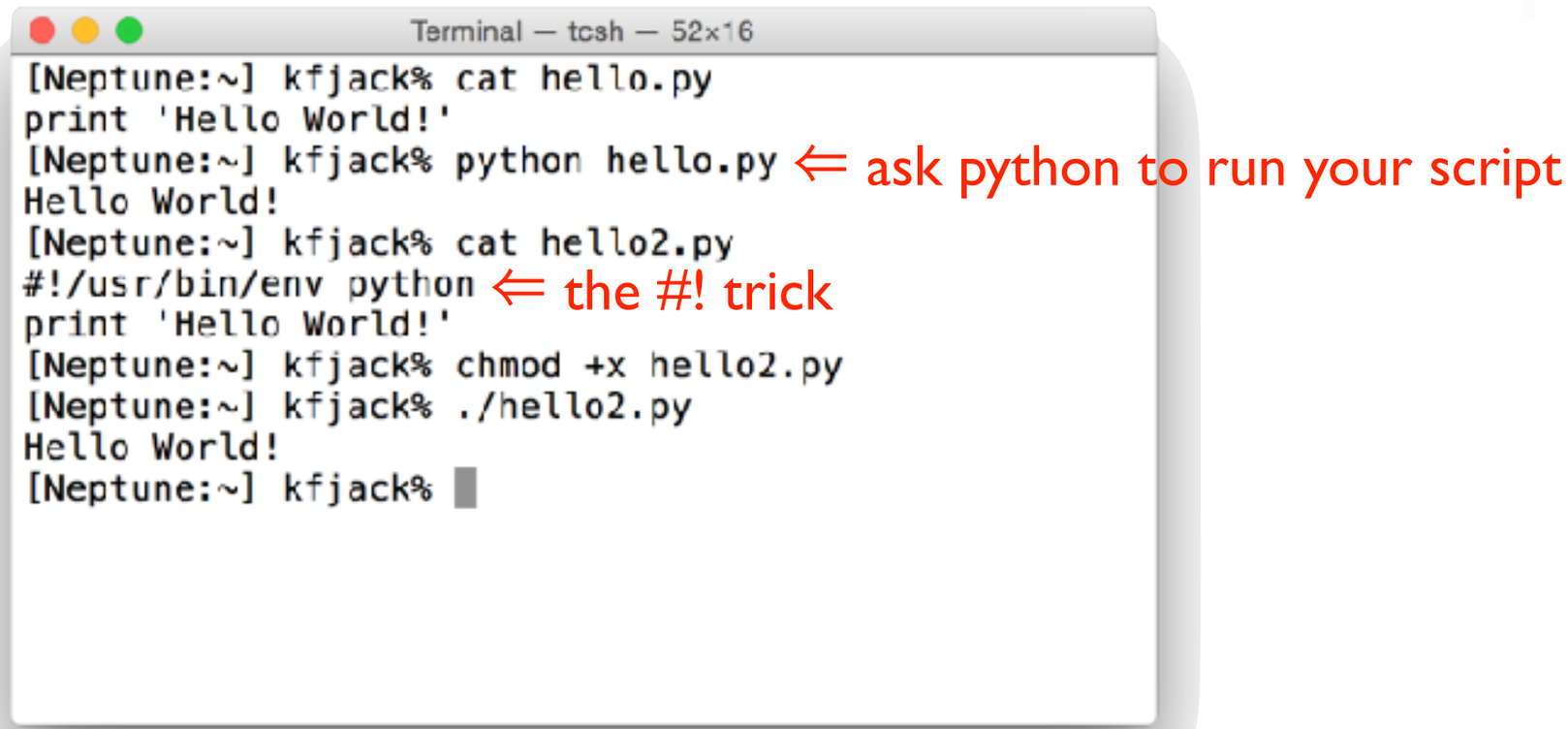
SOME COMMON ISSUES (CONT.)

- In the **interpreter mode**, the python interpreter can be used as a calculator, ie. if you type in any math operations it will show the answer on the screen directly.
- If you type in any **variable**, the value of the variable will be shown.
- If you type in any **function**, the return value of the function will be shown.

```
% python
>>> 2+2
4
>>> x = 123
>>> x  ⇐ this is a variable
123
>>> type(x)  ⇐ this is a function
<class 'int'>
```

SOME COMMON ISSUES (CONT.)

- **Issue #2: How to use the script mode** — put your code into a file (mostly named with .py, any text editor can be used!), and execute in the terminal command **"python xxx.py"**.
- if you use the Unix/Linux "#!" trick in your code, then you do not need to type in "python xxx.py", but just **"xxx.py"**, if you have added the executable permission to the file.



SOME COMMON ISSUES (CONT.)

- In the **script mode**, you cannot do what we did in interpreter mode. You cannot just type the math calculations, or variables, or functions to show the output. You have to use the **"print"** at least.

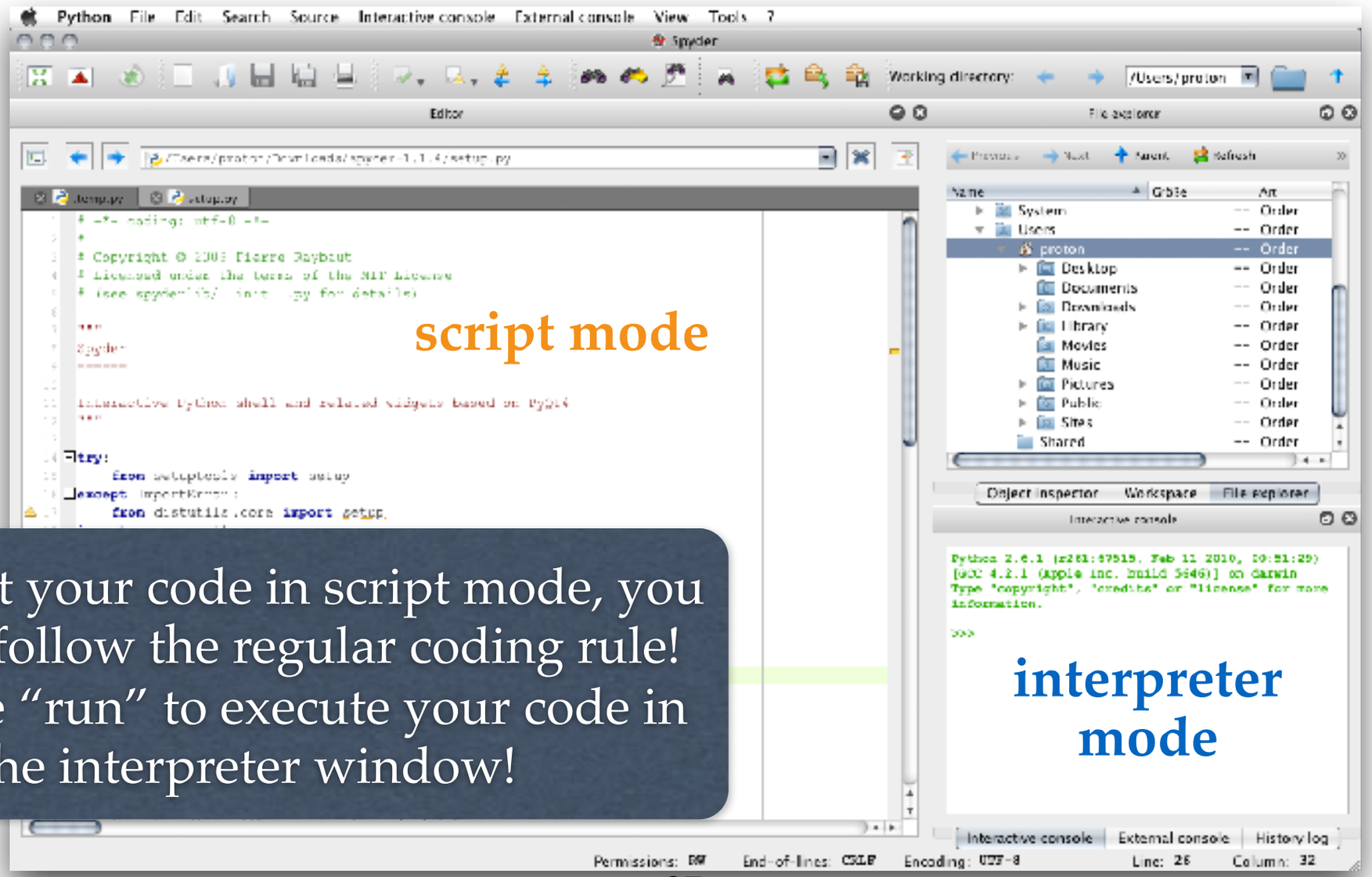
In the slides of this lecture, for the code in the script mode, it is always shown in a yellow box!

```
2+2 # this will not show anything
type(123) # this will not show anything

print(2+2) # this will print 4 on the screen
print(type(123)) # this will print <class 'int'> on the screen
```

SOME COMMON ISSUES (CONT.)

- **Issue #3: But I'm using an IDE from Anaconda / Canopy…** if this is the case, you have both "modes" available in a single screen:



If you put your code in script mode, you have to follow the regular coding rule! Press the "run" to execute your code in the interpreter window!

SOME COMMON ISSUES (CONT.)

- **Issue #4: Why I double-clicked my xxx.py, it just shows something quickly and disappeared quickly?** — You are likely using a windows system. The default action when a .py file is double-clicked will be just "run it". Since your program runs very fast, your program terminate and the command line window also closed right away, so you cannot see the output.
  - *Solution #1*: start a command line window by yourself and run the program by typing the commands.
  - *Solution #2*: always use your IDE to load your .py code and run.
  - *Solution #3*: add the following two lines to the end of your code:

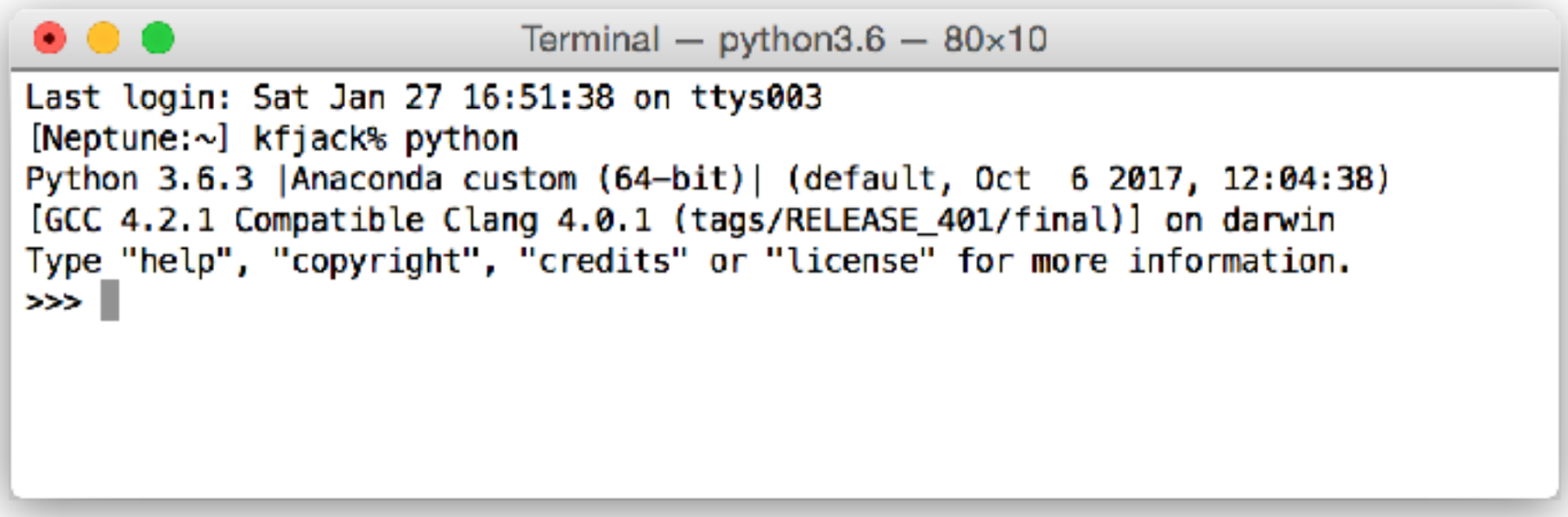
```
import msvcrt
msvcrt.getch() ⇐ this will wait for a key (Windows only!).
```

HANDS-ON SESSION

- Up to now we have gone through:
  - The basic python interface
  - Variables, types, and operators
- Now you should be able to do some useful calculations!
- Let's start our first round of hands-on session now!

HANDS-ON SESSION

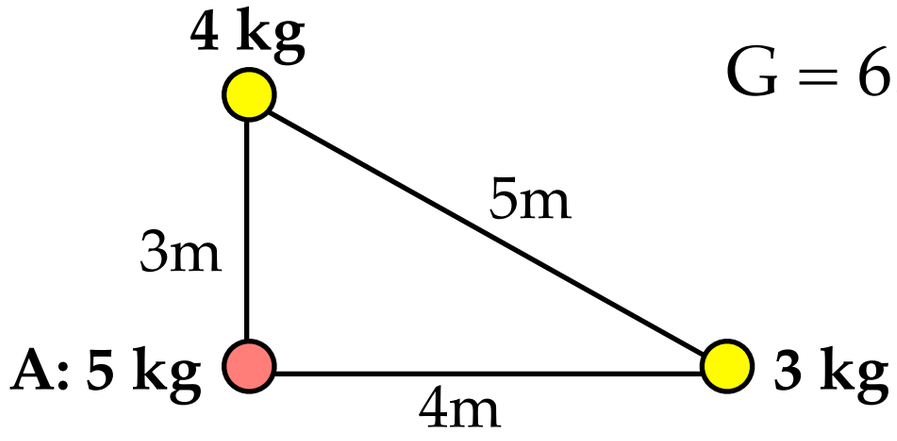
- **Practice 1:**
  Get your working environment ready now, and type in your first "hello world!".



HANDS-ON SESSION

- **Practice 2:**
  Use the python interpreter as a calculator, and calculate the following simple math problems:

  - Given $z = \dfrac{-2}{1+\sqrt{3}i}$, what's the real part and imaginary part?
  - Calculate the magnitude of the total gravitational force that is acted on point mass A:



$G = 6.67384 \times 10^{-11}\ m^3 kg^{-1} s^{-2}$

$$F = \frac{GM_1M_2}{R^2}$$