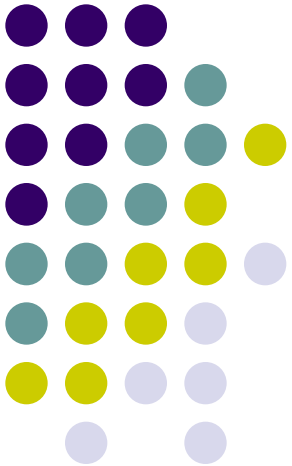
Tutorial 1: Introduction to Python

EE2405

嵌入式系統與實驗

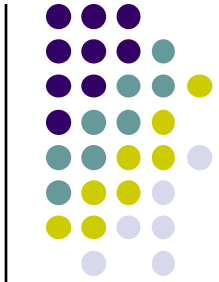
Embedded System Lab



Topics

- What is Python
- Python syntaxes
- Python programs

What is Python?



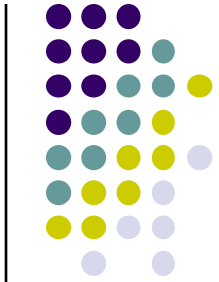
- Python: A free (open source), portable, dynamically-typed, object-oriented scripting language.
- Created in 1990 by Guido van Rossum
- Combines software engineering features of traditional systems languages with power and flexibility of scripting languages.

DOCTOR FUN

6 Apr 2000

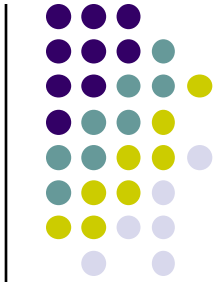
"Doctor Fun has the dubious distinction of being the first web cartoon.
Doctor Fun was not, however, the first cartoon on the Internet." - http://www.ibiblio.org/Dave/index.html

Python language features



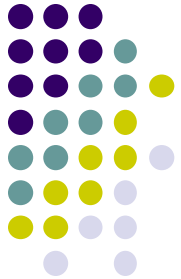
- Object-Oriented
- Dynamically typed
- Interpreted
- Cross-platform (Unix, Windows, etc.)
- Extensible
  - With C/C++, Java, etc.
- Powerful standard library
- Easy to use and maintain

The Python Standard Library



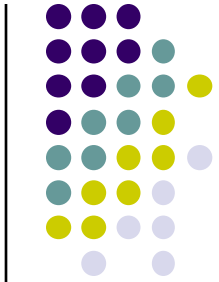
- GUI
- strings
- regular expressions
- database connectivity
- HTTP, HTML, XML
- numeric processing
- debugger
- object persistence
- OS services
- …

What is a "scripting" language?



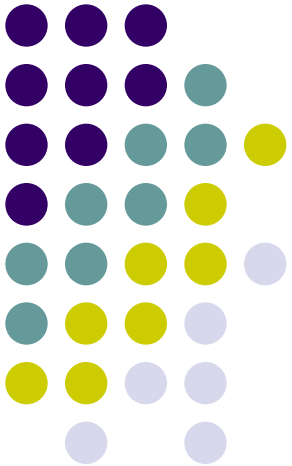
- Interpreted
  - requires a run-time interpreter or virtual machine
  - No compilation step
- Un-typed or dynamically typed
  - No data declarations

Online Materials



- Python distribution includes:
  - Tutorial, Language Reference
  - Extensive Standard Library documentation
  - http://www.python.org
- ”Think Python 2nd Ed" https://greenteapress.com/wp/think-python-2e/
  - A freely available Python book
- Too many others to list, please search python tutorial.

Introduction to Python Data Types



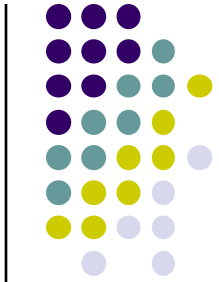
Starting Python in Linux

- **On Linux/Unix systems, Python may already be installed**

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

  - This starts the interpreter and allows you to type programs interactively.
  - Can press Control-D (Unix) or Control-Z (Windows) to exit interactive interpreter
  - Python3 is recommended over Python2
    - There is some incompatibility between Python3 and Python2

Variables and Types (1 of 3)

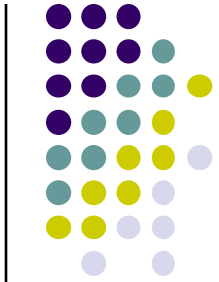


- Variables need no declaration
- `>>> a=1`
  `>>>`
- As a variable assignment is a statement, there is no printed result
- `>>> a`
  `1`
- Variable name alone is an expression, so the result is printed

Variables and Types (2 of 3)

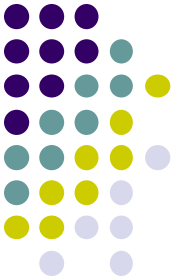
- Variables must be created before they can be used
- **>>> b**
  Traceback (innermost last):
  File "<interactive input>", line 1, in ?
  NameError: b
  >>>
- Python uses exceptions - more detail later

Variables and Types (3 of 3)



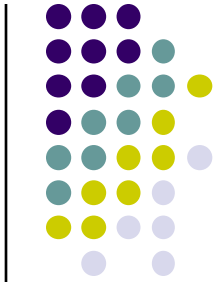
- Objects always have a type
- **>>> a = 1**
  >>> type(a)
  <type 'int'>
  >>> a = "Hello"
  >>> type(a)
  <type 'string'>
  >>> type(1.0)
  <type 'float'>

Assignment versus Equality Testing



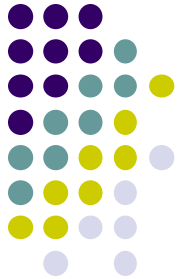
- Assignment performed with single =
- Equality testing done with double = (==)
  - Sensible type promotions are defined
  - Identity tested with `is` operator.
- **>>> 1==1**
  1
  >>> 1.0==1
  1
  >>> "1"==1
  0

Strings

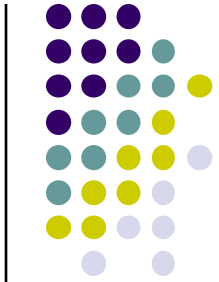


- Strings
  - May hold any data, including embedded NULLs
  - Declared using either single, double, or triple quotes
  - ```
    >>> s = "Hi there"
    >>> s
    'Hi there'
    >>> s = "Embedded 'quote'"
    >>> s
    "Embedded 'quote'"
    ```
    ```

Multi-line Strings



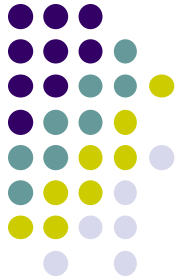
- Triple quotes useful for multi-line strings
- **>>> s = """ a long**
  ... string with "quotes" or anything else"""
  >>> s
  ' a long\012string with "quotes" or anything else'
  >>> len(s)
  45



Integer and Float

- Integer objects implemented using C longs
  - Unlike C, integer division returns the float
  - **>>> 5/2**
    2.5
- Float types implemented using C doubles
  - **>>> 5.0/2**
    2.5

Long Integer

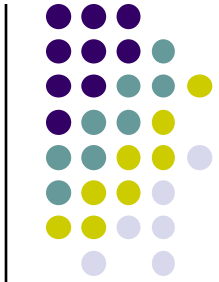


- Long Integers have unlimited size
  - Limited only by available memory
  - **>>> long = 1L << 64**
    >>> long ** 5
    2135987035920910082395021706169552114602704522356652769947041607822219725780640550022962086936576L

High Level Data Types --- List

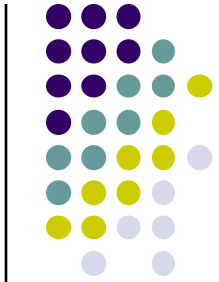
- Lists hold a sequence of items
  - May hold any object
  - Declared using square brackets
- **>>> l = []# An empty list**
  >>> l.append("Hi there")
  >>> l.append(”world")
  >>> len(l)
  2

List Examples



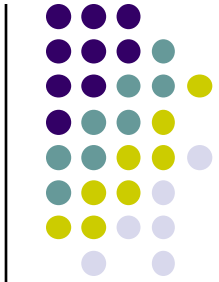
```
>>> l=[] # An empty list
>>> l.append(7)
>>> l.append(4)
>>> len(l)
2
>>> l
[7, 4]
>>> l=l+[6]
>>> l
[7, 4, 6]
>>> l.sort()
>>> l
[4, 6, 7]
>>>
```

Tuple



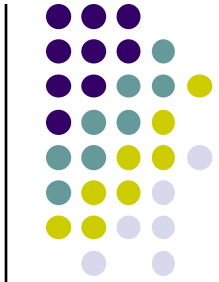
- Tuples are similar to lists
  - Sequence of items
  - Key difference is they are immutable
  - Often used in place of simple structures
- Automatic unpacking
- **>>> point = 2,3**
  >>> point = (2,3)
  >>> point[1]
  2
  >>> x, y = point
  >>> x
  2

Return with Tuples



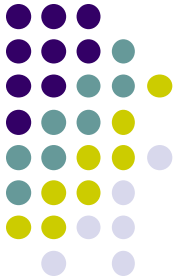
- Tuples are particularly useful to return multiple values from a function
- **>>> x, y = GetPoint()**
- As Python has no concept of reference parameters, this technique is used widely

Dictionary



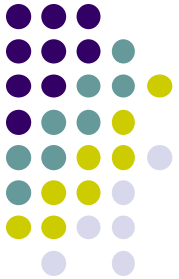
- Dictionaries hold key-value pairs
  - Often called maps or hashes. Implemented using hash-tables
  - Keys may be any immutable object, values may be any object
  - Declared using braces

Dictionary Examples



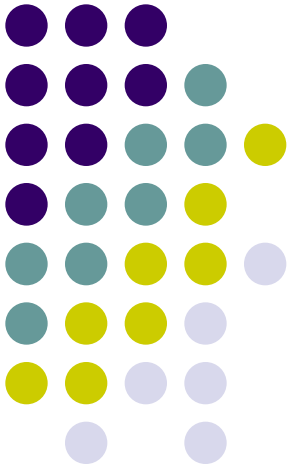
```
>>> d={}
>>> d["foo"]="Hi, there"
>>> d["bar"]="Hello"
>>> len(d)
2
>>> d
{'foo': 'Hi, there', 'bar': 'Hello'}
>>> d["bar"]
'Hello'
>>>
```

Dictionary Examples

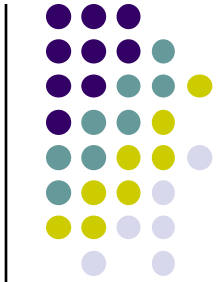


```
>>> d={}
>>> d['Mary']=['senior', 'EE']
>>> d['Peter']=['freshman', 'CS']
>>> d
{'Mary': ['senior', 'EE'], 'Peter': ['freshman', 'CS']}
>>> d['Mary'][1]
'EE'
>>>
```

Writing Python Programs



A Quick Example: First Program (C++ Version)



- Print "Hello NTHU EE" on screen
- In print.cc

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello NTHU EE\n";
    exit(0);
}
```

- **Compile and run**
  - **gcc –o print print.cc**
  - **./print**

First Program (Python Version)

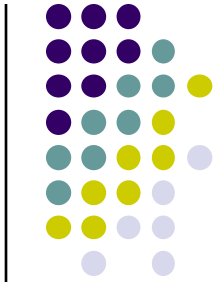
- Print "Hello NTHU EE" on screen
- print.py

```
print("Hello NTHU EE")
```

- Run

  $ python3 print.py

Running Python Programs



- Hybrid compiled/interpreted architecture
- Options:
  - Start Python with script file as command line arg
    - python3 script.py
  - Configure file type to launch interpreter on file
    - Unix script trick
    - #!/usr/bin/env python3
  - Start Interpreter from command line (>>>)
    - Type program statements
    - Import script file
  - Directly from IDE
    - ”Start debugging” in VSC

Setup a Python Script in Linux

- In hello.py

```
#!/opt/local/bin/python3
print("Hello NTHU EE")
```

- This the Unix #! trick
  - The first line “#!/usr/local/bin/python3” is to use python3 as the interpreter to run hello.py

- Make sure python3 path

  $ which python3

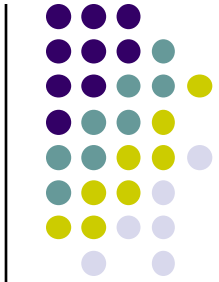
  /opt/local/bin/python3
- Make the script executable

  $ chmod +x ./hello.py
- Run the script

  $ ./hello.py

  Hello NTHU EE

Program Termination

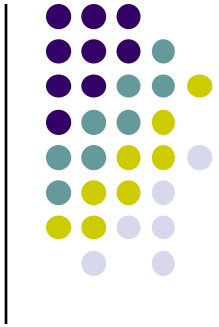


- Program Execution
  - Programs run until there are no more statements to execute.
  - Usually this is signaled by EOF
- Forcing Termination
  - Raising an exception:
    >>> raise SystemExit
  - Calling exit manually:
    import sys
    sys.exit(0)

A "Real" Program: Chaos.py

```
#!/usr/bin/env python
#File: chaos.py
# A simple program illustrating chaotic behavior.
def main():
      print("This program illustrates a chaotic
function")
      x = input("Enter a number between 0 and 1: ")
      for i in range(10):
            x = 3.9 * x * (1 - x)
            print(x)
main()
```

Notes on Chaos.py



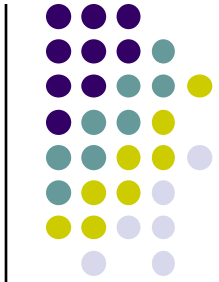
Comments begin with a #

Functions begin with a def

```
#!/usr/bin/env python
#File: chaos.py
# A simple program illustrating chaotic behavior.
def main():
    print("This program illustrates a chaotic function")
    x = input("Enter a number between 0 and 1: ")
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)
main()
```

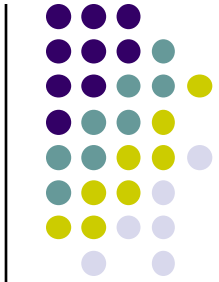
Note these spaces are meaningful in a Python. They define a new program structure. Here it defines the range of a “for”-loop

Notes on Chaos.py (Cont.)



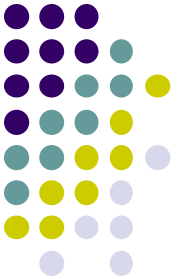
- Special built-in functions used by chaos.py
- input("message")
  - print the "message" on the screen and obtain a value typed in by the user (in the return value).
- range()
  - produce a "list" of numbers
  - >>>range(10)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Notes on Chaos.py (Cont.)



- Special structures used by chaos.py
- Python will run chaos.py line-by-line from the beginning to the EOF
- def main():
  - Define the function “main”
- Python will actually “run” main() after the construction of “main()” definition.
- for i in range(10):
  - Define a for-loop
  - “I” is the iteration variable by assigning “i” with one value from [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  - So the loop will run for 10 times.

Python Syntax Features Summary

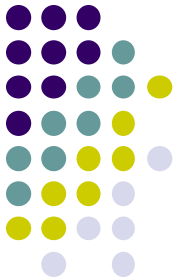


- Comments
  - "#" to end of line
- Nesting indicated by indentation
  - Count the number of leading “space chars”
- Statements terminated by end of line
- No variable declarations
  - Any used variable is created dynamically
- For loop iterates through a sequence of items

More on Code Blocks (Indent)

- Code blocks are delimited by indentation
  - Colon used to start a block
  - Have to use the same number of space characters to indent a block.
  - Tabs or spaces may be used
  - Maxing tabs and spaces works, but usually this is discouraged

More on Line Structure



- Line continuation (\) can be used for long statements

```
a = math.cos(3*(x-n)) + \
math.sin(3*(y-n))
```

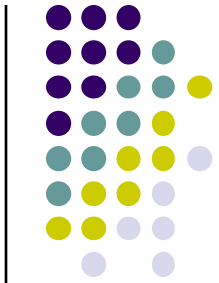
- Triple-quoted strings, lists, tuples, and dictionaries can span multiple lines

```
a = [ 4, 5 ]
b = { 'x': 10,
     'y': 20 }
c = foo(x,y,z,
        w,v)
```

- Short statements can be separated by a semicolon (;)

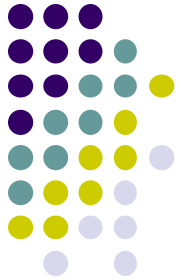
```
a = 3; b = 10*x; print b
```

Output of chaos.py



```
$ python chaos.py
This program illustrates a chaotic function
Enter a number between 0 and 1: .5
0.975
0.0950625
0.335499922266
0.869464925259
0.442633109113
0.962165255337
0.141972779362
0.4750843862
0.972578927537
0.104009713267
$
```

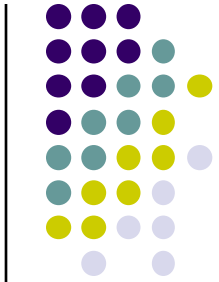
Document String for chaos.py



```
#!/usr/bin/env python
"""This function print out a chaotic function for 10
iterations"""
#File: chaos.py
# A simple program illustrating chaotic behavior.
def main():
      print("This program illustrates a chaotic
function")
      x = input("Enter a number between 0 and 1: ")
      for i in range(10):
            x = 3.9 * x * (1 - x)
            print(x)
main()
```

Now you can use help(chaos)!

Documentation Strings



- Documentation string
- Text string in the first statement of function, class, module, etc.
  - Example:

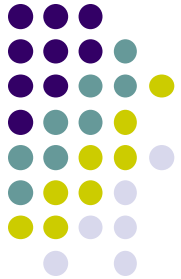
```
def factorial(n):
    """This function computes n factorial"""
    if (n <= 1): return 1
    else: return n*factorial(n-1)
```

- Accessing documentation strings
  - Found by looking at __doc__ attribute.

```
>>> print factorial.__doc__
This function computes n factorial
>>>
```

- Code browsers and IDEs often look at doc-strings to help you out.
- And it's considered to be good Python programming style.

Looping



- The `for` statement loops over sequences
- **>>> for ch in "Hello":**

  ... print(ch)

  ...

  H

  e

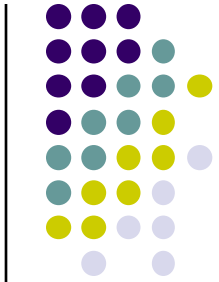
  l

  l

  o

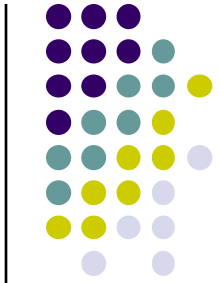
  >>>

Looping with range()



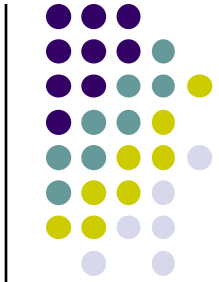
- Built-in function `range()` used to build sequences of integers
- **>>> for i in range(3):**
  ... print(i)
  ...
  0
  1
  2
  >>>

While Loop



- `while` statement for more traditional loops
-
```
>>> i = 0
>>> while i < 3:
...     print(i)
...     i = i + 1
...
0
1
2
>>>
```

Functions

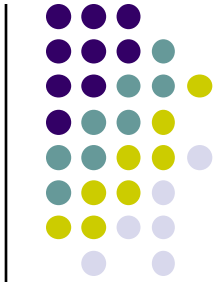


- Functions are defined with the **`def`** statement:
-
```
>>> def foo(bar):
...     return bar
>>>
```
- This defines a trivial function named **`foo`** that takes a single parameter **`bar`**

Function Example

```
def factorial(n):
    """This function computes n factorial"""
    if (n <= 1): return 1
    else: return n*factorial(n-1)
def main():
    j=10
    print('factorial of', j, '=',
factorial(j))
main()
```

Notes on Function Example

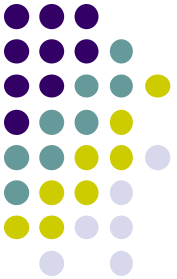


- function factorial()
  - Take one argument, n
  - Return one result
  - A recursive function

- Program output

```
$ python3 factorial.py
factorial of 10 = 3628800
```

Classes

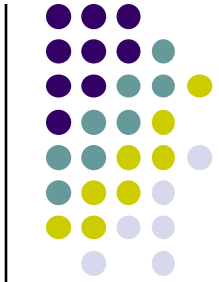


- Classes are defined using the `class` statement
-
```
>>> class Foo:
...     def __init__(self):
...         self.member = 1
...     def GetMember(self):
...         return self.member
...
>>>
```

Classes

- A few things are worth pointing out in the previous example:
  - The constructor has a special name **`__init__`**, while a destructor (not shown) uses **`__del__`**
  - The **`self`** parameter is the instance (ie, the **`this`** in C++). In Python, the self parameter is explicit (c.f. C++, where it is implicit)
  - The name **`self`** is not required - simply a convention

Classes



- Like functions, a class statement simply adds a class object to the namespace
- ```
  >>> Foo
  <class __main__.Foo at 1000960>
  >>>
  ```
- Classes are instantiated using call syntax
- ```
  >>> f=Foo()
  >>> f.GetMember()
  1
  ```

Class Example

```
import math

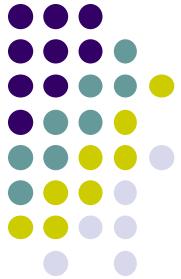
class circle():
    def __init__(self, r=1):
        self.r=r
    def GetArea(self):
        return math.pi*self.r*self.r

def main():
    circles=[circle(1), circle(2)]
    for c in circles:
        print(f'The area of circles[{circles.index(c)}] is
{c.GetArea():.2f}')

    for i in range(len(circles)):
        circles[i].r=circles[i].r+1
        print('The area of circles[{0}] is {1:.3f}'.format(i,
circles[i].GetArea()))

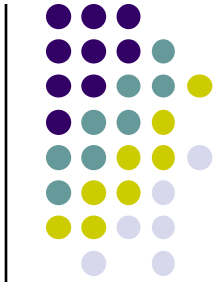
main()
```

Circle Example Output



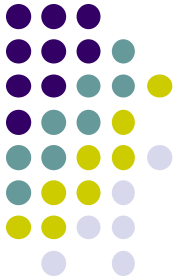
```
$ python3 circle.py
The area of circles[0] is 3.14
The area of circles[1] is 12.57
The area of circles[0] is 12.566
The area of circles[1] is 28.274
```

Notes on Circle



- class circle
  - __init__ defines the constructor
  - Default parameter r=1
  - Define an object variable self.r
  - Define a GetArea() method
- main()
  - Append two circle objects into a list
  - (first loop) Loop over all list of circles and print their area
  - (second loop) Loop over all list of circles using an index, add 1 to radius, and print their area

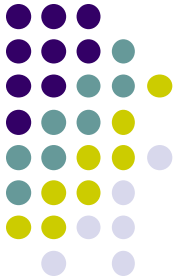
Notes on print Formats (1)



- f’ {} ‘
  - `f'The area of circles[{circles.index(c)}] is {c.GetArea():.2f}’`
  - Use {} to include a variable
  - Use : after a variable to indicate number of digits
  - .2f means to print a float with two digits after dot
  - 4d mean to print a int with 4 digits
  - index() is a method to search index value of a list

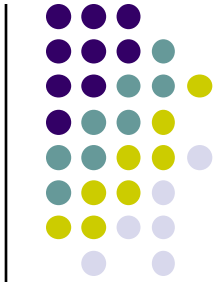
- ’ {0} ‘.format() form

Notes on print Formats (1)



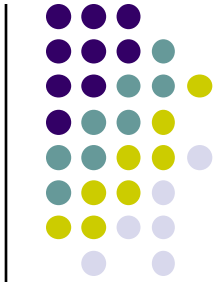
- ’ {0} ‘.format() form
  - `'The area of circles[{0}] is {1:.3f}'.format(i, circles[i].GetArea())`
  - .format() to list all used variables
  - Use {} in a string to include a variable
    - {0} means the first variable listed in format()
    - {1} means the second variable listed in format()
    - etc.
  - Again we can use : after a variable to indicate number of digits
  - {1:.3f} means to print the second variable as a float with three digits after dot

Modules



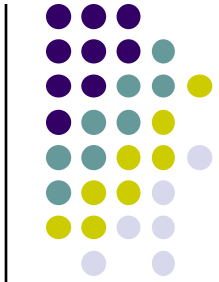
- Most of Python's power comes from modules
- Modules can be implemented either in Python, or in C/C++
- **`import`** statement makes a module available
-
```
>>> import string
>>> string.join( ["Hi", "there"] )
'Hi there'
>>>
```

Exceptions



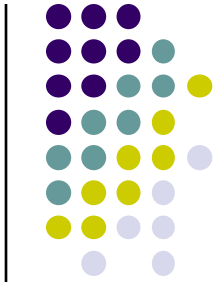
- Python uses exceptions for errors
  - `try` / `except` block can handle exceptions
- **>>> try:**
  ... 1/0
  ... except ZeroDivisionError:
  ... print("Eeek")
  ...
  Eeek
  >>>

Exceptions



- `try` / `finally` block can guarantee execute of code even in the face of exceptions
- **>>> try:**
  ... 1/0
  ... finally:
  ... print("Doing this anyway")
  ...
  Doing this anyway
  Traceback (innermost last): File "<interactive input>", line 2, in ?
  ZeroDivisionError: integer division or modulo
  >>>

Thread Example



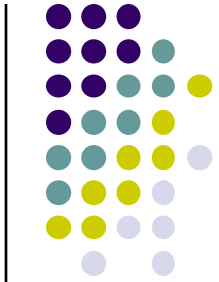
```
import threading
import time

class DemoThread(threading.Thread):
    def __init__(self, j, k):
        threading.Thread.__init__(self)
        self.j=j
        self.k=k
    def run(self):
        for i in range(self.j, self.k):
            time.sleep(0.5)
            print(i)

def main():
    t1 = DemoThread(1, 3)
    t1.start()
    t2 = DemoThread(4, 6)
    t2.start()
    t1.join()
    t2.join()

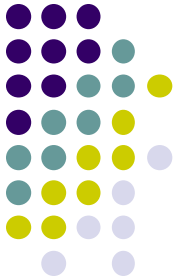
main()
```

Notes on Thread Example



- class DemoThread
  - Inherited from threading.Thread
  - Constructor calls threading.Thread constructor
  - Constructor setup two variables: j and k
  - Define run() which is the thread function
    - Count from j to k and print the counter
    - To be executed when the thread is invoked.
- main()
  - Define threads and start them
  - Wait for threads to complete by join()

Output of Thread Example



```
$ python3 thread.py
1
4
2
5
```