A Quick Tour of Python

Prof. Pai H. Chou
National Tsing Hua University

A Quick Tour of Python

- Input/Output
- Comments
- Operators, variables, assignments, numbers, strings
- Lists, Dictionaries, Tuples
- `if`, `while`, `for` loops, indentation
- Functions and Modules
- Object Oriented Programming

Input/Output (I/O)

- Basic assumption: text terminal
  - input: keyboard. use input() function in Python
  - output: text display. use print() function in Python
- Other I/O
  - file reading/writing: call built-in file functions
  - network send/receive: import `socket` module and call socket functions

Output: print()

- `print()` is a built-in function that displays a value as text output
  - works with numbers, strings, lists, ...
- Example

```
>>> x = 23
>>> print('there are', x, 'frogs')
there are 23 frogs
>>>
```

- Note:
  - parentheses are required (in Python 3 and later)
  - a space separates the items, a new line added at the end

Formatted Output

- Syntax:
  - `print('format string' % args to be formatted)`
  - the format can be `%d` for decimal, `%s` for string,..
- Example

```
>>> x = 23
>>> print('there are %d frogs' % x)
there are 23 frogs
>>> print('there are %04d frogs' % x)
there are 0023 frogs
>>> print('there are %4d frogs' % x)
there are   23 frogs
```

Reading input from the keyboard using the input() function

- Save the program as plain text file named `welcome.py`

```
x = input('What is your name? ')
print('Welcome, %s!' % x)
```

- Execute (run) this file from the command line interface:

```
$ python3 welcome.py
What is your name? Whitney
Welcome, Whitney!
```

Comments

- Line Comments
  - Start with the # character till end of line
  - works in both interactive and batch modes

```
# this is a comment
# this is a simple program for asking the user for name
# and say welcome to the user

x = input('What is your name? ')  # prompt user for name
print('Welcome, %s!' % x)
```

- Purpose: explain to humans
  - for anyone reading the code, including yourself!

Summary: input and output

- print() function
  - outputs text to the text display
  - displays different types of data (e.g., number) as text
- input() function
  - reads one line at a time from user keyboard
  - allows users to make correction before Return
  - option for displaying prompt

Operators in Python

- Arithmetic
- Logical
- Comparison
- String

Operators

- Mathematical
  - **+** (plus), **–** (minus), ***** (times), **/** (divide),
    // integer divide, **%** remainder, ****** exponentiation

```
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 7 % 3
1
>>> 7 ** 3
343
```

Operators

- Comparison
  - `<` (less than), `<=` (less than or equal to),
    `>` (greater than), `>=` (greater than or equal to),
    `==` equal to, `!=` not equal to
    - Result: `True` or `False`

```
>>> x = 2                    #  = is the assignment operator
>>> x < 1
False
>>> x == 2
True
```

Logical Operators

- a **`and`** b: True if both a and b are True
- a **`or`** b: True if either or both are True
- **`not`** a: True if a is False, False if a is True

```
>>> a = True
>>> b = False
>>> not a
False
>>> a and b
False
>>> a or b
True
```

Truth Table

a	b	a and b	a or b
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Numbers: integers

- Numerals in different bases
  - decimal (base 10), hexadecimal (base 16), octal (base 8)

dec	hex	octal	python decimal	python hex	python octal
0..7	0..7	0..7	0..7	0x0..0x7	0o0..0o7
8	8	10	8	0x8	0o10
9	9	11	9	0x9	0o11
10	A	12	10	0xA	0o12
11	B	13	11	0xB	0o13
12	C	14	12	0xC	0o14
13	D	15	13	0xD	0o15
14	E	16	14	0xE	0o16
15	F	17	15	0xF	0o17
16	10	20	16	0x10	0o20

Printing numbers in different bases

- by default, prints in decimal

```
>>> print(12, 0x12, 0o12)
12 18 10
```

- Format strings using
  - %x for hex,
  - %o for octal,
  - %d for decimal

```
>>> print('%d %x %o' % (12, 0x12, 0o12))
12 12 12
```

Floating point and Complex numbers

- Floating point
  - The following are equivalent floating point values
    `300.` `300.0` `3e2` `float(300)`
  - Difference from integers: division

```
>>> 3.0 / 2.0
1.5
>>> 3 / 2
1.5
```

```
>>> 3 // 2
1
>>> 3.0 // 2.0
1.0
```

- Complex number
  - example: `2+3j`

    `2` is the *real* component, `3j` is the *imaginary* component

    `j` is square-root of -1

Strings and the slice operator

- A string can be indexed from 0 to n-1
  - (0 up to **but not including** n, where n = length)
  - negative index means counting from the back

```
>>> x = 'Python'
>>> x[0]
'P'
>>> x[1:3]
'yt'
>>> x[2:]
'thon'
>>> x[-1]
'n'
>>> x[-3:-1]
'ho'
```

+ (concatenate) and * (repeat) operators for strings

```
>>> 'Hello' + ' ' + 'world!'
'Hello world!'
>>> '-' * 20
'--------------------'
>>> 'abc' * 5
'abcabcabcabcabc'
```

Summary: Operators

- Operators are "punctuation marks" that perform a function
- Different operators in Python
  - Arithmetic
  - Logical
  - Comparison
  - String

Collection Data Types

- Lists　(列表)
- Tuples (元組)
- Dictionaries (字典)
- Sets（集合)

Lists

- sequence of values, indexable from 0

```
>>> L = [1, 2, 3]
>>> L[0]
1
```

- Lists are mutable (can be modified)

```
>>> L[1] = 'Hi'
>>> L
[1, 'Hi', 3]
```

- can concatenate (+) and repeat (*) nondestructively

```
>>> L + [6, 7, 8]
[1, 'Hi', 3, 6, 7, 8]
>>> L
[1, 'Hi', 3]
```

Tuples

- similar to arrays, using `( )` instead of `[ ]`

```
>>> T = (12, 34, 56)
>>> T[2]
56
```

  - actually, comma is the operator for making tuples!
- but indexing uses `[ ]` also, just like lists
- Tuples are "read-only" (immutable)
  - i.e., cannot say `T[1] = 'Hi'` if `T` is a tuple
    (but it is OK to modify a list)

Dictionaries

- Unordered key-value pairs
- Allows lookup by name
  - name could be numbers, strings, tuples (i.e., an immutable value)
  - syntax: use **{ }** to enclose key:value pairs

```
>>> d = {'Jan': 1, 'Feb': 2, 'Mar': 3 }
>>> d['Feb']
2
>>> d['Apr'] = 4    # can add more key:value pairs to a dictionary
```

Sets

- Unordered collection of values
  - use `{ }` or `set()` to construct a set
- Operators
  - `-` (set subtraction), `|` (union), `&` (intersection), `^` (exclusive-union = their difference)

```
>>> A = { 1, 2, 3 }; B = { 1, 3, 5 }; C = { 2, 4, 6 }
>>> A & B
{1, 3}
>>> A | C
{1, 2, 3, 4, 6}
>>> A - B
{2}
>>> A ^ B
{2, 5}
```

Summary: Collection Data Types

- Data structure that consists of multiple other parts.
  - Sequences: lists [1,2,3], tuples (1,2,3)
  - Unordered: dictionaries {1:2, 3:4}, sets {1,2,3}
- Purpose
  - store multiple data items in a container
  - access items by position, by key, or membership

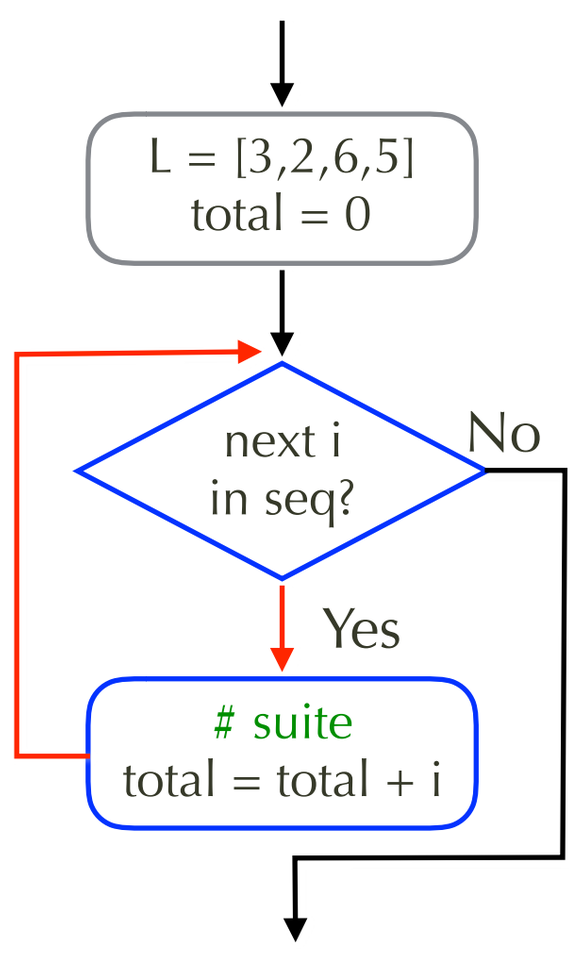
Control-Flow Constructs

- `for` loop
- `if` statement
- `while` loop

for-loop

- Repeats over each element of a sequence
  - a loop variable is updated each time
- Example: add up a sequence of numbers

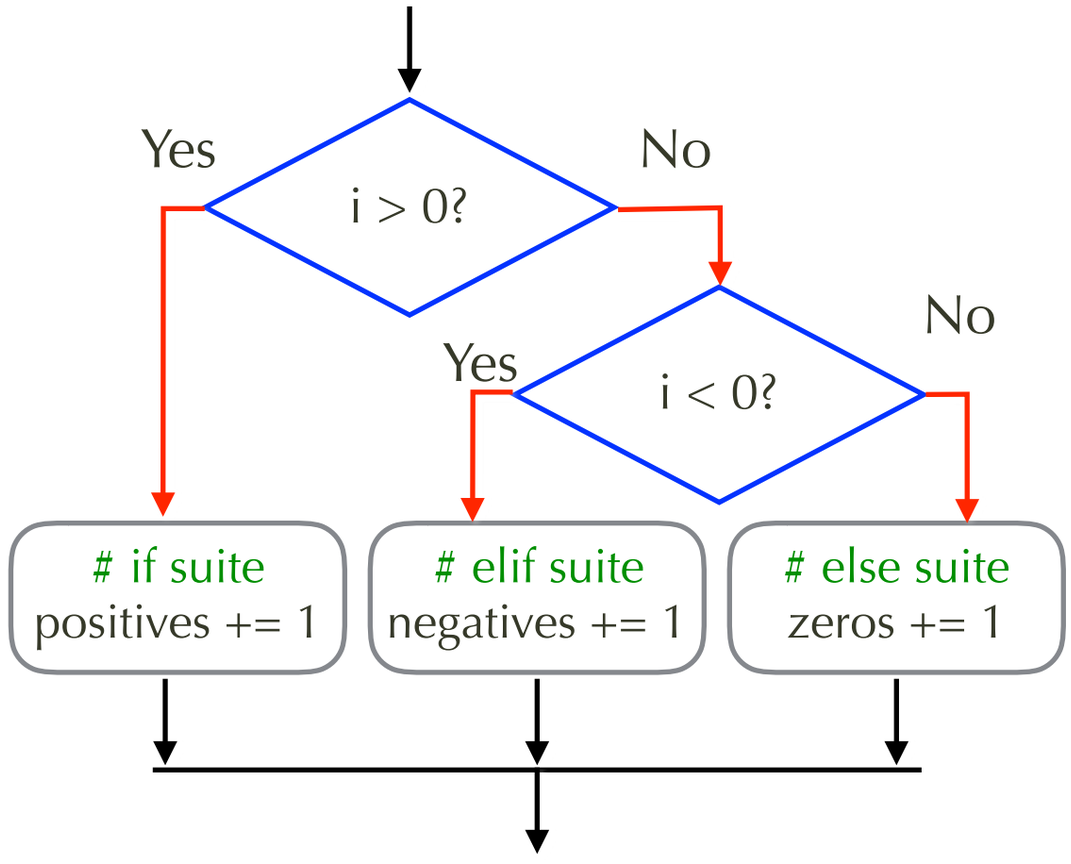
```
>>> L = [3, 2, 6, 5]
>>> total = 0
>>> for i in L:
...     total += i  # total = total + i
...
>>> total
16
```



`if`-statement

- conditional execution
- example: count how many negatives, positives, and zeros are in a list

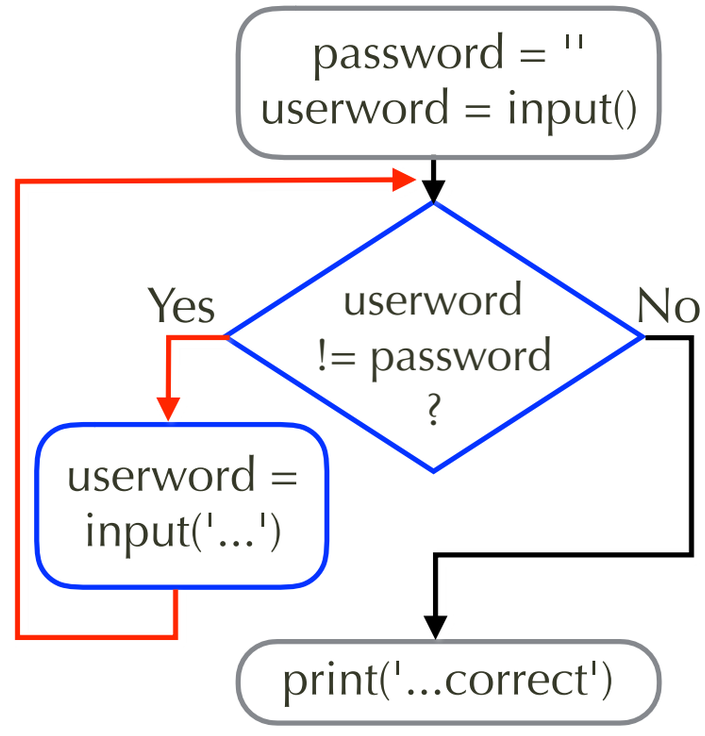
```
L = [5, -9 -6, 8, 0, 7, 4]
negatives, positives, zeros = 0, 0, 0
for i in L:
    if i > 0:
        positives += 1
    elif i < 0:
        negatives += 1
    else:  # i can only be == 0
        zeros += 1
print('>0: %d, <0: %d, ==0: %d' % \
      (positives, negatives, zeros))
```



`while-loop`

- repeats as long as the condition is true
  - unlike for-loop, we don't necessarily know in advance how many times the loop will run
- Example: asks the user for password until it matches

```
password = 'ABCDE'
userword = input('Password: ')
while (userword != password):
    userword = input('Wrong passwd, try again: ')
print('Your password is correct')
```



Summary: Control constructs

- for loop
  - repeat the loop over a sequence of items
- if statement
  - executes the body conditionally
- while loop
  - repeat as long as condition evaluates to true
- "suite"
  - indented statement block inside a control construct

Functions and Modules

- Functions
  - Defining and calling
  - Indentation
- Modules
  - importing library and your own
  - import vs. from .. import ..

Calling functions

- calling predefined functions
- e.g., `input("your name: ")`
  - name of function is `input`
  - argument passed to the call is `"your name: "`
  - the function call returns the string typed by user
- e.g., `print("x = ", x)`
  - parameters are string "x = " and value of x.
  - this function call does not return a value (returns `None`)

Define your own function

- Make your code reusable by
  - defining parameters, instead of hardwiring values
  - return values
- Example: convert "total" code to function

```
>>> L = [3, 2, 6, 5]
>>> total = 0
>>> for i in L:
...     total += i
...
>>> total
16
```

define

```
def Total(L):
    total = 0
    for i in L:
        total += i
    return total
```

call

```
x = Total([3,2,6,5])
```

```
print(Total([3,2,6,5]))
```

Documentation String for functions

- The first string after def serves as documentation for the function.
  - useful as mini-documentation

```
def twice(x):
    'This returns x plus itself'
    return x + x
```

save in file named "myfile.py"

```
$ python3 -i myfile.py # run and enter interactive mode
>>> help(twice)
Help on function twice in module myfile:

twice(x)
    This returns x plus itself
(END)
>>>
```

run file with -i flag to enter interactive mode

Modules

- Reusable collection of code and definition
- Bring them in by `import`

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(0)
1.0
>>> math.cos(math.pi)
-1.0
```

- `help(math)` to get documentation

Many useful Modules

- basics
  - datetime, time
  - random
  - sys, os

```
>>> import random
>>> L = list(range(5))
>>> L
[0, 1, 2, 3, 4]
>>> random.shuffle(L)
[0, 2, 1, 3, 4]
```

- bundled or 3rd party
  - audio, image, video, scientific plotting
  - user interface, gaming
  - web service, database

See https://wiki.python.org/moin/UsefulModules

Your own Python files are also modules

- You can save Python code in a file
  - e.g., myfile.py

```
def twice(x):
    'This returns x plus itself'
    return x + x
```

- Two ways to import

```
>>> import myfile          # import myfile into __main__ namespace
>>> myfile.twice(2)
4
>>> from myfile import * # import everything in myfile into __main__
>>> twice(3)
6
```

Summary: functions and modules

- function
  - a subprogram that can be reused by calling
  - call by name, pass parameter, get return value
- module
  - a collection of functions and variables
  - can be library that you import or your own file

More for-loops: range, list, files

- For-loop with `range()`
- For-loop with list
- File open
- File with list of lines

Recap: for-loop

- Repeats over each element of a sequence
  - a loop variable is updated each time
- Example: add up a sequence of numbers

```
>>> L = [3, 2, 6, 5]
>>> total = 0
>>> for i in L:
...     total = total + i
...
>>> total
16
```

range(): Systematic way of generating the sequence of numbers

- Instead of `[0, 1, 2, 3, 4]` in for loop, easier to say `range(5)`
  - means from 0 up to but not including 5
  - `range(5)` *generates* the sequence 0, 1, 2, 3, 4
  - can make a list by `list(range(5))`

```
>>> total = 0
>>> for i in range(5):
...     total += i
...
>>> total
10
```

More ways of using range()

- Can specify
  - start (inclusive; default = 0)
  - stop (exclusive; required)
  - step (default = 1)

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

file access: reading

- open a file using `open()`
  - need to give it a file name
  - reading (default) or writing (need to specify)
  - open returns a file handle
- Use the file handle to read/write, close
  - why? several files might be open at the same time, need to know which one you want to access!
- Open could fail!
  - e.g., file not found, no permission
  - Python response: Raising an *exception*

Example: Count the number of lines in file "myfile.txt"

```
>>> h = open('myfile.txt', 'r')  # open file for reading
>>> L = h.readlines()  # read the whole file into L as lines
>>> L[0]               # display the first line, if it exists
>>> h.close()          # close the file
>>> len(L)             # length of L is the number of lines
```

Summary: Loops, Files

- `range()` built-in function
  - generates sequence of integers
  - doesn't take space for all numbers in range
- For-loop with list, range(), or generator
- File open
  - File reading into a list of lines
  - use for loop to walk the list of lines

Object-Oriented Programming

- object = data with function ("methods")
  - in Python, all data are objects
- Example

```
>>> x = 'hello'
>>> x.upper()           # example of method call on string object!
'HELLO'
>>> L = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
>>> L.index('Tue')      # find index of value 'Tue'
2
```

Object-oriented programming

- Class vs. instance
  - Class: definition for data and code ("blueprint")
  - Instance: object made based on class definition
- Instantiation
  - Creating an instance of a class
- Method invocation
  - Calling a function of an object to process attributes associated with the data

Example class and instance

```
class Student:      # class declaration
    def __init__(self, name, ID):   # constructor
        self.name = name
        self.ID = ID
    def showMe(self):
        print('Name:', self.name, 'ID:', self.ID)
    def changeName(self, newName):
        self.name = newName
```

```
% python -i student.py   # run the file and stay in interactive mode
>>> a = Student('Donald Trump', 12345) # a is an instance of Student
>>> a.showMe()
Name: Donald Trump ID: 12345
>>> a.changeName('Barak Obama')
>>> a.showMe()
Name: Barak Obama ID: 12345
>>> b = Student('George Bush', 67890) # b is another instance
```

Summary of object-oriented programming

- General way to define data type
  - data created by instantiation of a class
  - "attribute": member data of an object
  - "method": member function invoked on object
- Purpose
  - good way to organize data and program
  - inheritance allows new class to be defined by specifying difference => less likely to make mistakes

Statements in Python

- Simple vs. Compound Statements
- Delimiters
- Multi-line string literals
- Assignment statements
- Multiple Assignments

Simple vs. Compound Statements

- Simple statements

```
i = x + f(y)  # delimited by newline
```

  - assignment or function call
  - Normally must fit on same line, except extended

- Compound statement:
  - starts w/colon (:) followed by indented suite
  - suite ends by restored indentation level

```
if x > 0:
    a = 1
    b = 2  # indented in if
```

b is assigned to 2
only if x > 0

```
if x > 0:
    a = 1
b = 2    # same level as if
```

b is assigned to 2
no matter what

Delimiter for simple statements

- Newline
  - can have optional comment

```
a = 1   # my comment
b = 2
```

- Semicolon
  - terminates one statement, begin next statement at the same level of indentation (not counting leading white spaces)

```
a = 1; b = 2
```

\ to extend a statement or string to multiple lines

- Use \ for extending statement on next line
  - why? so that your long code can fit in a limited number of columns (80 columns is standard)

```
f = a + b * 2 + c / 2 \
    - 4 * d
```

- \ works with strings too:

```
s = 'This is a string that \
is extended to the next line'
```

  - without the \ it would be a syntax error!

Multi-line string literal

- Use triple-quotes
  - either triple-single or triple-double quotes
  - allows a string to contain newlines!
  - Can still use \ for line extension

```
>>> a = """Hi,
...  there"""
>>> print(a)
Hi
 there
>>>
```

```
>>> a = """Hi, \
...  there"""
>>> print(a)
Hi  there
>>>
```

Assignment statements

- variable = expression
- expression can involve
  - constant:

```
i = 1
s = 'Hello'
```

```
L = [3.56, -2.7]
t = (1, 2, 3)
```

  - variable

```
i = j
s = s[1:3]
```

```
t = (i, 2, j)
L[i] = s
```

  - return value of a function call

```
h = open('test.py', 'r')
```

```
s = reverse('abcd')
```

  - operator
  - and their combinations

```
z = x + y ** 2
b = (x < 1) and (y > 3)
L = [1, 2] + [x, y]
y = (a - b * sin(x)) / 2
```

Multiple Assignment

- variable = variable = ... = expression

```
x = y = z = 1
```

```
S1 = S2 = 'hello'
```

  - constant can appear only at the far right
  - not ok =>

```
x = 1 = y
```

    why not? because 1 is not mutable.

- packed assignment

```
(a, b, c) = (1, 2, 3)
```

  - evaluates tuple first, then assign

Swapping numbers

- Can use packed assignment for swapping

```
(x, y) = (y, x)
```

- Without packed assignment
  - => use a temporary variable

```
temp = x  # save the value of x before overwriting it
x = y     # now safe to overwrite x
y = temp  # y gets the saved value of x
```

- Incorrect to do

```
x = y   # x does get y, but old value of x is lost!
y = x   # y gets the new value of x, which is y!
```

Summary: Statements in Python

- Simple: line-delimited
  - function call, assignment
  - use \ to extend to multiple lines
  - triple-quoted strings may go on multiple lines
- Compound statements
  - contain suite (indented code block)
- Assignment
  - single, multiple, and packed

Identifiers, Keyword

- Identifiers
- Keywords
- Case sensitivity
- Coding Style

Identifiers vs. Keywords

- Keywords
  - words with reserved meaning in Python language
  - cannot be reused for other purposes!
- Identifiers
  - all other words that can be defined to mean
    - a variable, a function, a module, a class, ...
  - can be redefined to mean something else!

Keywords in Python

- **if**, **for**, **def**, **class**, **import**, **while**, **return**, ...
  - they are special, like operators **+**, **–**, *, /, **, (,), ...

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

- Error if you try to use a keyword as a variable, a function, a class, etc.

```
>>> in = 20
  File "<stdin>", line 1
    in = 20
     ^
SyntaxError: invalid syntax
```

Identifiers in Python

- Rules
  - must start with a letter or an underscore (_)
  - may be followed by letters, digits, or underscores
  - case sensitive
- Built-in identifiers
  - sys, open, input, ...
- Reserved identifiers (special meaning)
  - `__doc__`, `__class__`, `__name__`, `__dict__`, ...

Examples of identifiers

- Legal
  - `A`, `a`, `x`, `y`, `YY`, `zZZZz`, `B12`, `m2t`, `_A`, `_`, `the_best`, `theBest`
- illegal
  - `$twenty`, `#people`, `red-shoes`, `ranked%`, `two+more`
  - `if`, `while`, `return`, (all are reserved word)
- Legal identifier, but avoid
  - `print`, `input` # predefined but not keywords!

Case sensitivity

- upper and lower case letters are different!
  - `RETURN` is ok as identifier (not a keyword)
- Examples of different identifiers:
  - `averagePower`, `AveragePower`, `AVERagePOWer`, `average_power`, `avErAgePoWeR`, ...
- Need to make sure you use the exact case for your identifier

Camel Case vs. Snake Case

- Two ways of making multi-word identifier
- Camel case: capitalize a new word
  - `averageMidtermScore`, `yearOfProduction`
  - think camel back going up and down
- Snake case: use _ to join words
  - `average_midterm_score`, `year_of_production`
  - think snake, thin and long
- Be consistent in your program, don't mix

Coding Style

```
#!/usr/bin/env python
"""documentation string for this
   module"""
import sys
import ... # fill in actual module name

debug = 1
class Student:
   ... # fill in class definition

def test():
   ... # code to test code

if __name__ == '__main__':
    test()
```

specify the python interpreter to use (in Unix)

documentation

other modules needed

global variables

class (data structure) definitions

function definition

code to test this module if invoked by itself (i.e., when not imported by another Python program)

Choose meaningful identifiers

- Limit the use of abbreviated ones
  - `i`, `j`, `k` permissible as generic index, but not very meaningful for general variable
  - Algebra variables `x`, `y`, `z`, `a`, `b`, `c` may be too generic, unless equation is already well known $e = m\ c^2$
- In general, use self-explanatory identifiers
  - `minWeight`, `avgCost`, `longestPath`, etc.

Summary: Identifiers, Keyword

- Identifiers
  - names that can be defined by programmer for their own meaning
- Keywords
  - words with reserved meaning in Python language and cannot be changed
- Coding Style
  - import, global, class, function, main