AI 프로그래밍

류광렬 지음

부산대학교 정보컴퓨터공학부 교수

Contents

01 Basics of Python

Values and Types

- Values belong to different types

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
>>> type(3.2)
<type 'float'>
>>> type('17')
<type 'str'>
>>> type((1, 0, 0))
>>> <type 'tuple'>
>>> 1,000,000
(1, 0, 0)
```

  - Python interprets 1,000,000 as a sequence of integers separated by commas → an example of a semantic error

Numbers

- Integers:

`2` `34` `-34`

- Floats (floating point numbers):

`34.` `3.23` `52.3E-4` $(= 52.3 \times 10^{-4})$

- The built-in functions **`abs`**, **`int`**, and **`round`**:

Expression	Value	Expression	Value	Expression	Value
`abs(3)`	3	`int(2.7)`	2	`round(2.7)`	3
`abs(0)`	0	`int(3)`	3	`round(2.317,2)`	2.32
`abs(-3)`	3	`int(-2.7)`	-2	`round(2.317,1)`	2.3

Variables

- Variables are just parts of our computer's memory where you store some information
  - We need some method of accessing these variables and hence we give them names
- An assignment statement creates new variable and gives them values

```
speed = 50
timeElapsed = 14
distance = speed * timeElapsed
print(distance)
```

[RUN]

```
700
```

- Naming rule:
  - Case sensitive: `myname` and `myName` are different
  - Must begin with a letter or an underscore (_)
  - The rest can be any of the alphabet, underscores, or digits
  - Descriptive variable names help others (and yourself) easily recall what the variable represents
- Examples of invalid identifier names:

```
2things
```

```
this is spaced out
```

```
my-name
```

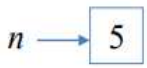
```
>a1b2_c3
```

- Python's keywords cannot be used as variable names
  - Python 2 has 31 keywords

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	

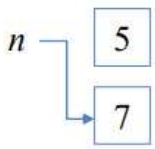
- Numeric objects in Memory:
  - Consider the following lines of code

```
n = 5
n = 7
```



✓ A portion of memory is set aside to hold 5

✓ The variable n is set to reference (or point to) 5 in the memory location



✓ A new memory location is set aside to hold 7

✓ The variable n is redirected to point to the new memory location

  - The number 5 in memory is said to be orphaned or abandoned
  - Python will eventually remove the orphaned number from memory with a process called garbage collection

Strings

- Strings are immutable
  - Cannot be changed once created
- A string is a sequence of characters
  - Single quotes: `'Quote me on this'`
  - Double quotes: `"What's your name?"`
  - Triple quotes (""" or ''') can be used to specify multi-line strings in which single quotes and double quotes can be used freely

```
>>> x = '''This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."'''
>>> x
'This is the first line.\nThis is the second line.\n"What\'s
your name?," I asked.\nHe said "Bond, James Bond."'
>>> print(x)
This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
```

Concatenation and Repetition

- Two strings can be concatenated by using the + operator
- A string can be repeatedly concatenated by using the * operator

```
>>> 'good' + 'bye'
'goodbye'
>>> ('a' + 'b') * 3
'ababab'
>>> 'a' * 3 + 'b' * 3
'aaabbb'
>>> 'ha' * 3
'hahaha'
>>> ("cha-" * 2) + "cha"
'cha-cha-cha'
```

- When a string expression appears in an assignment statement or a print statement, the string expression is evaluated before being assigned or displayed

String Functions and Methods

```
>>> len('string')
6
>>> int('23')
23
>>> float('23')
23.0
>>> eval('23')
23
>>> eval('23.5')
23.5
>>> x = 5
>>> eval('23 + (2 * x)')
33
```

- The **`eval`** function evaluates the expression to an integer or floating-point number as appropriate
- The **`exec`** function takes a string consisting of Python code and executes it

```
>>> exec('x = 2')
>>> x
2
>>> exec('y = 3')
>>> y
3
>>> eval('x + y')
5
>>> exec('x + y')
>>>
```

- The **`int`** and **`float`** functions can also be applied to numeric expressions

Example	Value	Example	Value
`int(4.8)`	`4`	`float(4.67)`	`4.67`
`int(-4.8)`	`-4`	`float(-4)`	`-4.0`
`int(4)`	`4`	`float(0)`	`0.0`

- The **`str`** function converts a number to its string representation

```
>>> str(5.6)
'5.6'
>>> str(5)
'5'
>>> str(5.)
'5.0'
>>> x = 10
>>> str(x) + '%'
'10%'
```

- A string method is a process that performs a task on a string
  - The general form of an expression applying a method is

    `stringName.methodName()`

    where the parentheses might contain values

```
>>> str1 = "Python"
>>> str1.upper()
'PYTHON'
>>> str1.lower()
'python'
>>> str1.count('th')
1
>>> 'coDE'.capitalize()
'Code'
>>> "beN hur".title()
'Ben Hur'
>>> 'ab   '.rstrip()  # removes spaces from the right side
'ab'
```

- String operations (**`str1 = "Python"`**)

Function or Method	Example	Value	Description
len	len(str1)	6	number of characters in the string
upper	str1.upper()	"PYTHON"	uppercases every alphabetical character
lower	str1.lower()	"python"	lowercases every alphabetical character
count	str1.count('th')	1	number of non-overlapping occurrences of the substring
capitalize	"coDE".capitalize()	"Code"	capitalizes the first letter of the string and lowercases the rest
title	"beN hur".title()	"Ben Hur"	capitalizes the first letter of each word in the string and lowercases the rest
rstrip	"ab ".rstrip()	"ab"	removes spaces from the right side of the string

- Chained methods:

```
>>> praise = "Good Doggie".upper()
>>> numberOfGees = praise.count('G')
>>> print(numberOfGees)
3
```

  - These two lines can be combined into a single line by chaining the two methods

```
>>> numberOfGees = "Good Doggie".upper().count('G')
>>> print(numberOfGees)
3
```

  - Chained methods are executed from left to right
  - Chaining often produces clearer code since it eliminates temporary variables, such as the variable **`praise`** above

Indices and Slices

- If **`str1`** is a string variable, then **`str1[i]`** is the character of the string having index i (the index starts from 0)

- A slice of a string is a sequence of consecutive characters from the string
  - **str1[m:n]** is the substring beginning at position *m* and ending at position *n* − 1
  - **str1[m:n]** will be the empty string ("") if $m \geq n$
  - Given another string **subStr**, the methods **str1.find(subStr)** and **str1.rfind(subStr)** return the positive index from the left and right, respectively, of the first appearance of **subStr** in **str1**
  - −1 is returned if **subStr** does not appear in **str1**

```
print('Python')
print('Python'[1], 'Python'[5], 'Python'[2:4])
str1 = 'Hello World!'
print(str1.find('W'))
print(str1.find('x'))
print(str1.rfind('l'))     # finds the rightmost 'l'
```

[RUN]

```
Python
y n th
6
-1
9
```

- Python allows strings to be indexed by their position from the right side of the string by using negative numbers for indices

```
print('Python')
print('Python'[-1], 'Python'[-4], 'Python'[-5:-2])
str1 = 'spam & eggs'
print(str1[-2])
print(str1[-8:-3])
print(str1[0:-1])
```

```
[RUN]

Python
n t yth
g
m & e
spam & egg
```

- One or both of the bounds in **`str1[m:n]`** can be omitted
  - *m* defaults to 0
  - *n* defaults to the length of the string

```
print('Python'[2:], 'Python'[:4], 'Python'[:])
print('Python'[-3:], 'Python'[:-3])

[RUN]

thon Pyth Python
hon Pyt
```

Optional print Arguments

- We can optionally change the separator with **`sep`** argument:

```
>>> x = 5; y = 7
>>> print(x, y, sep='*')
5*7
>>> print("Hello", "World", sep="")
HelloWorld
>>> print('1', 'two', 3, sep='    ')
1    two    3
```

- **`print`** always ends with an invisible special character "new line" (**`\n`** or **`₩n`**) so that repeated calls to print will all appear on a separate new line each
- We can optionally change the ending operation with **`end`** argument:

```
print("Hello", end=" ")
print("World")
```

[RUN]

```
Hello World
```

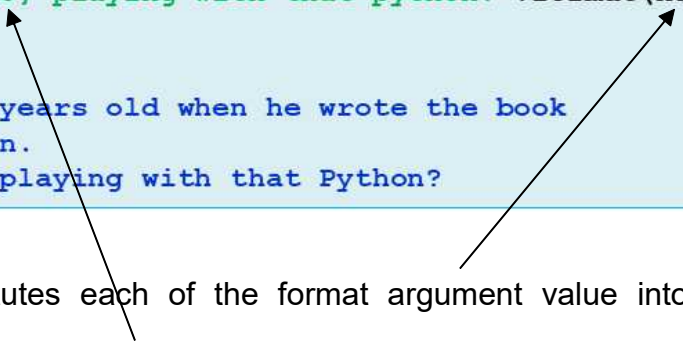
The format Method

- Strings can be constructed from other information

```
age = 20
name = 'Swaroop'

print('''{0} was {1} years old when he wrote the book
A Byte of Python.'''.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

[RUN]

```
Swaroop was 20 years old when he wrote the book
A Byte of Python.
Why is Swaroop playing with that Python?
```



  - Python substitutes each of the format argument value into the place of the corresponding specification in the string
  - Note that Python starts counting from 0

- Also note that the numbers in the specifications are optional
  - The following code gives exactly the same output as the previous code

```
age = 20
name = 'Swaroop'

print('''{} was {} years old when he wrote the book
A Byte of Python.'''.format(name, age))
print('Why is {} playing with that python?'.format(name))
```

[RUN]

```
Swaroop was 20 years old when he wrote the book
A Byte of Python.
Why is Swaroop playing with that Python?
```

- The symbols <, ^, and > that precede the width of each field instruct the print function to left-justify, center, and right-justify, respectively

```
## Demonstrate justification of output.
print("01234567890123456789012345678")
print("{0:^5}{1:<20}{2:>3}".format("Rank", "Player", "HR"))
print("{0:^5}{1:<20}{2:>3}".format(1, "Barry Bonds", 762))
print("{0:^5}{1:<20}{2:>3}".format(2, "Hank Aaron", 755))
print("{0:^5}{1:<20}{2:>3}".format(3, "Babe Ruth", 714))
```

[RUN]

```
01234567890123456789012345678
Rank  Player                HR
  1   Barry Bonds          762
  2   Hank Aaron           755
  3   Babe Ruth            714
```

  - When none of the symbols <, ^, or > are present, the number (string) will be displayed left-justified (right-justified) by default

- `f` and `%` are used after the field-width number to display a floating-point number or a number in percentages, respectively
  - They should be preceded by a period and a number indicating the decimal precision
  - A comma can be inserted after the field-width number if we want thousands separators

Statement	Outcome
print('{0:10.2f}'.format(1234.5678))	1234.57
print('{0:10,.2f}'.format(1234.5678))	1,234.57
print('{0:10,.3f}'.format(1234.5678))	1,234.568
print('{0:10,.2%}'.format(1234.5678))	123,456.78%
print('{0:10,.3%}'.format(1234.5678))	123,456.780%
print('{0:10,}'.format(12345678))	12,345,678

- More on the format method

```
# decimal (.) precision of 3 for a float
print('{0:.3f}'.format(1.0/3))
# fill with underscores (_) with the text centered (^)
# to the width of 11
print('{0:_^11}'.format('hello'))
# keyword-based specifications
print('{name} wrote {book}'.format(name = 'Swaroop',
                                   book = 'A Byte of Python'))
```

[RUN]

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

  - Note: `1.0/3` is a float division

    `1/3` is an integer division resulting in `0`

Escape Sequences

- How can you specify a string that has a single quote in it?

```
>>> print('What's your name?')

SyntaxError: invalid syntax
```

```
print("What's your name?")
print('What\'s your name?')
print('He said, "Bond, James Bond."')
print("He said, \"Bond, James Bond.\"")
```

```
[RUN]
What's your name?
What's your name?
He said, "Bond, James Bond."
He said, "Bond, James Bond."
```

* '\' actually appears as '₩' in Python windows

- Escape sequences are short sequences that are placed in strings to permit some special characters to be printed
  - The first character is always a backslash (\)
- A backslash itself can be specified by using an additional backslash

```
print('How can you prevent \\n from being printed?')
print('How can you prevent \n from being printed?')
print('A backslash at the end of the line \
indicates line continuation')
[RUN]
How can you prevent \n from being printed?
How can you prevent
 from being printed?
A backslash at the end of the line indicates line continuation
```

- To specify some strings where no special processing such as escape sequences are handled
  - Specify a raw string by prefixing **`r`** or **`R`** to the string

Keyboard Input

- When a built-in function called **`input`** is called, the program stops and waits for the user to type something
  - When the user presses *Enter*, the program resumes and **`input`** returns what the user typed as a string

```
>>> text = input()
What are you waiting for?
>>> print(text)
What are you waiting for?
```

- If you want to print a prompt telling the user what to input, you can give the prompt to **input** as an argument

```
>>> name = input('What is your name? ')
What is your name? Allen
>>> print(name)
Allen
>>> name
'Allen'
```

- If you expect the user to type an integer, you can try to convert the return value to **int**

```
>>> prompt = 'What is the airspeed velocity of a swallow?\n'
>>> speed = int(input(prompt))
What is the airspeed velocity of a swallow?
17
>>> speed
17
```

  - The user's input appears below the prompt because the new line character **\n** at the end of the prompt causes a line break

```
fullName = input('Enter a full name: ')
n = fullName.rfind(' ')
print('Last name:', fullName[n+1:])
print('First name(s):', fullName[:n])
```

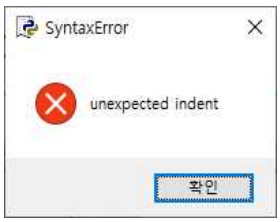
[RUN]

```
Enter a full name: Franklin Delano Roosevelt
Last name: Roosevelt
First name(s): Franklin Delano
```

Indentation and Line Joining

- Indentation is semantically meaningful in Python
  - Statements that go together (called a block) must have the same indentation
  - An indentation must have four spaces
- Wrong indentation gives rise to errors

```
i = 5
# Error below! Notice a single space at the start of the line
 print 'Value is ', i
print 'I repeat, the value is ', i
```



- Explicit line joining
  - A long logical line can be broken down to multiple physical lines by using the backslash

```
print('The area of {0} is {1:,} square miles.'
      .format('Texas', 268820))
str1 = 'The population of {0} is {1:.2%} of \ # continue to
the U.S. population.'                          # the next line
print(str1.format('Texas', 26448000. / 309000000))
```

[RUN]

```
The area of Texas is 268,820 square miles.
The population of Texas is 8.56% of the U.S. population.
```

- Implicit line joining
  - Backslash is not needed when the logical line has a starting parentheses, starting square brackets, or a starting curly braces but not an ending one

```
quotation = ('Well written code is its own ' +
             'best documentation.')
print(quotation)
```

[RUN]

```
Well written code is its own best documentation.
```

02 Operators and Control Flow

Operators

- Operators do something on operands

```
>>> 3 + 5  # Plus
8
>>> 3 + 5.0
8.0
```

```
>>> 50 - 24  # Minus
26
>>> -5.2
-5.2
```

```
>>> 2 * 3  # Multiply
6
>>> 2.0 * 3
6.0
```

```
>>> 3 ** 4  # Power
81
```

```
>>> 15 / 3   # Float division
5.0
>>> 13 // 3  # Integer division resulting in an integer
4
>>> 3.5 // 2 # Integer division resulting in a float
1.0
```

```
>>> 13 % 3  # Modulo (returns the remainder of the division)
1
>>> -25.5 % 2.25
1.5
```

* $-25.5 = 2.25 \times (-12) + 1.5$

```
>>> 2 << 2  # Left Shift
8
```

* Left shifting 10 by 2 bits gives 1000

```
>>> 11 >> 1  # Right Shift
5
```

* Right shifting 1011 by 1 bit gives 101

```
>>> 5 & 3  # Bitwise AND
1
```

* 101 AND 011 = 001

```
>>> 5 | 3  # Bitwise OR
7
```

* 101 OR 011 = 111

```
>>> 5 ^ 3  # Bitwise XOR
6
```

* 101 XOR 011 = 110

```
>>> ~ 5  # Bitwise Invert (bit-wise inversion of x is -(x+1))
-6
```

* Bit-wise inversion of 0101 is 1010, which is a two's complement representation of -6

```
>>> 5 < 3  # Less Than
False
>>> 3 < 5
True
>>> 3 < 5 < 7
True
```

```
>>> 5 > 3  # Greater Than
True
```

```
>>> x = 3
>>> y = 6
>>> x <= y  # Less Than or Equal To
True
```

```
>>> x = 4
>>> y = 3
>>> x >= y  # Greater Than or Equal To
True
```

```
>>> 2 == 2  # Equal To
True
>>> 'str' == 'stR'
False
>>> 'str' == 'str'
True
```

```
>>> 2 != 3  # Not Equal To
True
```

```
>>> x = True
>>> not x  # Boolean Not
False
```

```
>>> x = False
>>> y = True
```

```
>>> x and y  # Boolean AND
False
```

* `x` and `y` returns `False` if `x` is `False`, else it returns the evaluation of `y`

(`y` is not evaluated if `x` is `False` -- short-circuit evaluation)

```
>>> x = False
>>> y = True
>>> x or y  # Boolean OR
True
```

* `x` or `y` returns `True` if `x` is `True`, else it returns the evaluation of `y`

(`y` is not evaluated if `x` is `True` -- short-circuit evaluation)

```
>>> l = 10
>>> m = 5
>>> n = 0
>>> (n != 0) and (l == (m / n))
False
```

* An error will occur if both parts of the compound condition are evaluated

- Relational operators

Python Notation	Numeric Meaning	String Meaning
==	equal to	identical to
!=	not equal to	different from
<	less than	precedes lexicographically
>	greater than	follows lexicographically
<=	less than or equal to	precedes lexicographically or is identical to
>=	greater than or equal to	follows lexicographically or is identical to
in		substring of
not in		not a substring of

- Methods that return either Boolean values

Method	Returns True when
str1.isdigit()	all of *str1*'s characters are digits
str1.isalpha()	all of *str1*'s characters are letters of the alphabet
str1.isalnum()	all of *str1*'s characters are letters of the alphabet or digits
str1.islower()	*str1* has at least 1 alphabetic character and all of its alphabetic characters are lowercase
str1.isupper()	*str1* has at least 1 alphabetic character and all of its alphabetic characters are uppercase
str1.isspace()	*str1* contains only whitespace characters

Shortcut for Math Operation and Assignment

- The shortcut expression for **`var = var operation expression`** is

  `var operation= expression`

  - The following are equivalent

```
>>> a = 2
>>> a = a + 3
```

```
>>> a = 2
>>> a += 3
```

Precedence and Order of Evaluation

- The following table summarizes the rules for precedence of all
  - Rows are in order of increasing precedence
  - Operators on the same line have the same precedence

`lambda`	Lambda Expression
`if - else`	Conditional Expression
`or`	Boolean OR
`and`	Boolean AND
`not`	Boolean NOT
`in, not in, is, is not, <, <=, >, >=, !=, ==`	Comparisons
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<, >>`	Shifts
`+, -`	Addition and Subtraction
`*, /, //, %`	Multiplication, Division, Integer Division, and Modulus
`+x, -x, ~x`	Positive, Negative, bitwise NOT
`**`	Exponentiation

`x[index], x[index:index], x(arguments…), x.attribute`	Subscription, slicing, call, attribute reference
`(expressions…), [expressions…], {key: value…}, {expressions…}`	Binding or tuple display, list display, dictionary display, set display

- It is better to use parentheses to group operators and operands appropriately in order to explicitly specify the precedence

Associativity

- Operators are usually associated from left to right
  - E.g., `2 + 3 + 4` is evaluated as `(2 + 3) + 4`
- Some operators like assignment operators have right to left associativity
  - E.g., `a = b = c` is treated as `a = (b = c)`

Expressions

- An expression is a combination of values, variables, and operators
  - A value all by itself is considered an expression, and so is a variable
- A statement is a unit of code that the Python interpreter can execute
  - We have seen two kinds of statement: print and assignment
- Technically an expression is also a statement
  - The important difference is that an expression has a value; a statement does not

```
>>> x = 7
>>> 17
17
>>> x
7
```

```
>>> x + 17
24
```

```
length = 5
breadth = 2

area = length * breadth

print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

[RUN]

```
Area is 10
Perimeter is 14
```

- Notice that Python puts a space between **`'Area is'`** and the variable **`area`** in the output

Decision Structures

- **`if`** and nested **`if-else`** statement

```
number = 23
guess = int(input('Enter an integer: '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
else:
    if guess < number:
        print('No, it is a little higher than that')
    else:
        print('No, it is a little lower than that')

print('Done')
# This last statement is always executed
# after the if statement is executed.
```

- The string entered by the user is converted to an integer using **`int()`** and then stored in the variable **`guess`** (assuming that the string contains a valid integer in the text)
- A colon at the end of **`if`** statement indicates to Python that a block of statements follows
- Notice that we use indentation levels to tell Python which statements belong to which block

```
[RUN]

Enter an integer : 50
No, it is a little lower than that
Done

[RUN]

Enter an integer : 22
No, it is a little higher than that
Done

[RUN]

Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

- The following code is equivalent to that of the previous example

```
number = 23
guess = int(input('Enter an integer: '))

if guess == number:
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
elif guess < number:
    print('No, it is a little higher than that')
else:
    print('No, it is a little lower than that')

print('Done')
```

- This extension of the **`if-else`** statement allows for more than two possible alternatives with the inclusion of **`elif`** clauses
  - Reduces the amount of indentation required
- The **`elif`** and **`else`** parts are optional

```
## Find the largest of three numbers.
firstNumber = eval(input('Enter the first number: '))
secondNumber = eval(input('Enter the second number: '))
thirdNumber = eval(input('Enter the third number: '))

# Determine and display the largest value.
largest = firstNumber
if secondNumber > largest:
    largest = secondNumber
if thirdNumber > largest:
    largest = thirdNumber
print('The largest number is', str(largest) + '.')
```

[Run]

```
Enter the first number: 3
Enter the second number: 7
Enter the third number: 4
The largest number is 7.
```

  - The integer stored in **`largest`** is converted to a string by **`str()`** before it is concatenated with another string (a period) by **`+`**
- A space will be printed before the period if we change the print statement as follows

```
. . . . . .
print('The largest number is', largest, '.')
```

[Run]

```
. . . . . .
The largest number is 7 .
```

The while Loop

- A while loop has the form

  while *condition*:

  indented block of statements

  - Python first checks the truth value of ***condition***
  - If the condition evaluates to **False**, Python skips over the body of the loop and continues to execute the optional else-block and then continues to the line (if any) after the loop
  - If the continuation condition evaluates to **True**, the body of the loop is executed
  - After each pass through the loop, the continuation condition is rechecked and the body will be continually executed until the condition evaluates to **False**

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer: '))
    if guess == number:
        print('Congratulations, you guessed it.')
        running = False  # this causes the while loop to stop
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

[Run]

```
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

- There is no need to repeatedly run the program for each guess, as we have done with the **`if-else`** example at the beginning

- If there is an **`else`** clause for a while loop, it is always executed unless you break out of the loop with a **`break`** statement
- The **`True`** and **`False`** are called Boolean types and they are equivalent to the values **`1`** and **`0`**, respectively

```
## Find the minimum, maximum, and average of a sequence of
## integers.

count = 0  # number of nonnegative integers input
total = 0  # sum of the nonnegative integers input

# Obtain numbers and determine count, min, and max.
print('(Enter -1 to terminate entering numbers.)')
num = int(input('Enter a nonnegative integer: '))
min = num
max = num

while num != -1:
    count += 1
    total += num
    if num < min:
        min = num
    if num > max:
        max = num
    num = int(input('Enter a nonnegative integer: '))
```

```
# Display results.
if count > 0:
    print('Minimum:', min)
    print('Maximum:', max)
    print('Average:', total / count)
else:
    print('No nonnegative integers were entered.')
```

[Run]

```
(Enter -1 to terminate entering numbers.)
Enter a nonnegative integer: 3
Enter a nonnegative integer: 7
Enter a nonnegative integer: 4
Enter a nonnegative integer: -1
Minimum: 3
Maximum: 7
Average: 4.66666666667
```

The for Loop

- The for loop is used to iterate through a sequence of objects

  for *var* in *sequence*:

  indented block of statements

  - The loop variable ***var*** is successively assigned each value in the sequence and the loop body is executed after each assignment
  - ***sequence*** might be an arithmetic progression of numbers, a string, a list, a tuple, or a file object
    - ◯ When *m* and *n* are integers such that *m* < *n*, we can use the function **list(range(*m*, *n*))** to generate

      [*m*, *m* + 1, *m* + 2, . . . , *n* − 1]
    - ◯ **range(*m*, *n*, *s*)** takes the step count of s instead of 1 (When *m* > *n*, *s* can be a negative integer)

◯ **range(0, *n*)** can be abbreviated to **range(*n*)**

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

[Run]

```
1
2
3
4
The for loop is over
```

▪ The **else** part is optional

◯ When included, it is always executed once after the for loop is over unless a **break** statement is encountered

- Nested for loops:

```
## Display a triangle of asterisks.
numberOfRows = int(input('Enter a number from 1 to 20: '))

for i in range(numberOfRows):
    for j in range(i + 1):
        print('*', end='')
    print()
```

[Run]

```
Enter a number from 1 to 20: 5
*
* *
* * *
* * * *
* * * * *
```

- Looping through the characters of a string:

```
## Reverse the letters in a word.
word = input('Enter a word: ')
reversedWord = ''
```

```
for ch in word:
    reversedWord = ch + reversedWord
print('The reversed word is ' + reversedWord + '.')
```

[Run]

```
Enter a word: zeus
The reversed word is suez.
```

- Looping through the lines of a text file:

```
infile = open('fileName.txt', 'r')

for line in infile:

    indented block of statements

infile.close()
```

  - First statement establishes connection between program and file (assuming the file is in the same folder as the program)
  - **`for`** loop reads each line of the file in succession
    - ◯ **`line`** contains each line as a string
    - ◯ Executes indented block of statement(s) for each line
  - Last statement terminates the connection

```
## Display presidents with a specified first name.
firstName = input('Enter a first name: ')
foundFlag = False
infile = open('USPresidents.txt', 'r')
for line in infile:
    if line.startswith(firstName + ' '):
        print(line.rstrip())
        foundFlag = True
infile.close()
if not foundFlag:
    print('No president had the first name', firstName + '.')
```

```
[Run]
Enter a first name: John
John Adams
John Quincy Adams
John Tyler
John F. Kennedy
```

- **`startswith`** is a string method
- The variable **`foundFlag`** tells us if at least one president had the requested first name
- The **`rstrip`** method removes the newline character at the end of each line of the text file
  - ◯ No matter how many there are, whitespaces (spaces, new line characters, tabs) are removed from the right side of a string

- The **`pass`** statement:
  - The header of a for loop must be followed by an indented block of at least one statement
  - The **`pass`** statement is a do-nothing placeholder statement

```
## Display the last line of a text file.
infile = open('USPresidents.txt', 'r')
for line in infile:
    pass
print(line.rstrip())
infile.close()

[Run]

Barack Obama
```

The break Statement

- The break statement causes an exit from anywhere in the body of a loop terminating the loop, even if the loop condition has not become **`False`** or the sequence of items has not been completely iterated over
- If you break out of a **`for`** or **`while`** loop, any corresponding loop **`else`** block is not executed

```
print('Enter QUIT to terminate entering something')
while True:
    s = input('Enter something : ')
    if s == 'QUIT':
        break
    print('Length of the string is', len(s))
print('Done')
```

[Run]

```
Enter QUIT to terminate entering something
Enter something : Programming is difficult
Length of the string is 24
Enter something : Use Python!
Length of the string is 11
Enter something : QUIT
Done
```

- The built-in **`len()`** function returns the length of the input string

The continue Statement

- The continue statement causes Python to skip the rest of the statements in the current loop body and to continue to the next iteration of the loop

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
```

```
    if len(s) < 3:
        print('Too short')
        continue
    print('Input is of sufficient length')
```

[Run]

```
Enter something : a
Too short
Enter something : 12
Too short
Enter something : abc
Input is of sufficient length
Enter something : quit
```

03 Data Structures

List

- A list is an ordered sequence of Python objects of any type
  - The objects do not have to all be the same type
- A list is constructed by writing its items enclosed in square brackets, with the items separated by commas, e.g.,

```
['Seahawks', 2014, 'CenturyLink Field']

[5, 10, 4, 5]

['spam', 'ni']
```

- Lists are usually assigned to a name

```
>>> team = ['Seahawks', 2014, 'CenturyLink Field']
>>> nums = [5, 10, 4, 5]
>>> words = ['spam', 'ni']
```

- Once created, you can add, remove, or search for items in the list
- A list is a mutable data type
- List operations

Function or Method	Example	Value	Description
len	len(words)	2	number of items in list
max	max(numbers)	10	greatest (items must have same type)
min	min(numbers)	4	least (items must have same type)
sum	sum(nums)	42	total (items must be numbers)
count	nums.count(5)	2	number of occurrences of an object
index	nums.index(4)	2	index of first occurrence of an object
reverse	words.reverse()	["ni", "spam"]	reverses the order of the items
clear	team.clear()	[]	[] is the empty list
append	nums.append(7)	[5, 10, 4, 5, 7]	inserts object at end of list
extend	nums.extend([1, 2])	[5, 10, 4, 5, 1, 2]	inserts new list's items at end of list
del	del team[−1]	["Seahawks", 2014]	removes item with stated index
remove	nums.remove(5)	[10, 4, 5]	removes first occurrence of an object
insert	nums.insert(1, "wink")	["spam", "wink", "ni"]	insert new item before item of given index
+	['a', 1] + [2, 'b']	['a', 1, 2, 'b']	concatenation; same as ['a', 1].extend([2,'b'])
*	[0] * 3	[0, 0, 0]	list repetition

- Like the characters in a string, items in a list are indexed from the front with positive indices starting with 0, and from the back with negative indices starting with –1
- The value of the item having index **`i`** can be changed with a statement of the form **`listName[i] = newValue`**

```
>>> words = ['spam', 'ni']
>>> words[1] = 'eggs'
>>> words
['spam', 'eggs']
```

Objects and Classes

- When you write **`i = 5`**, you are creating an object **`i`** of class **`int`**
  - Use **`help(int)`** to get more information about **`int`** class
- A class can have methods, i.e., functions defined for use with respect to that class only, e.g.,

  `mylist.append('an item')`

  will add the string **`'an item'`** to the end of the list **`mylist`**
  - We can add any objects to a list, even other lists
- Some methods do things without returning any value

```
>>> a = [3, 2, 4, 1]
>>> print(a.sort())
None
>>> a
[1, 2, 3, 4]
```

```
shoplist = ['apple', 'mango', 'carrot', 'banana']

print('I have {0} items to purchase.'.format(len(shoplist)))
```

```
print('These items are:', end='')
for item in shoplist:
   print(item, end='')

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My list is now', shoplist)

print('I will sort my shopping list now')
shoplist.sort()
print('Sorted list is', shoplist)

print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

[Run]

```
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my shopping list now
Sorted list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

- The **append** and **sort** methods for the list class actually alters the list
- The **del** statement removes the specified item form the specified object

- More list functions and methods are shown below:

```
>>> team = ['Seahawks', 2014, 'CenturyLink Field']
>>> nums = [5, 10, 4, 5]
>>> words = ['spam', 'ni']
>>> max(nums)
10
>>> min(nums)
4
```

```
>>> sum(nums)
24
>>> nums.count(5)
2
>>> nums.index(4)
2
>>> words.reverse()
>>> words
['ni', 'spam']
>>> nums.append(7)
>>> nums
[5, 10, 4, 5, 7]
>>> nums.extend([1,2])
>>> nums
[5, 10, 4, 5, 7, 1, 2]
>>> nums.remove(5)  # removes first occurrence of an object
>>> nums
[10, 4, 5, 7, 1, 2]
>>> del nums[1:3]
>>> nums
[10, 7, 1, 2]
>>> words.insert(1,'wink')  # before item of given index
>>> words
['ni', 'wink', 'spam']
>>> ['a',1] + [2,'b']
['a', 1, 2, 'b']
>>> [0] * 3
[0, 0, 0]
```

- The **`split`** method turns a single string into a list of substrings

```
>>> 'a,b,c'.split(',')
['a', 'b', 'c']
>>> 'a**b**c'.split('**')
['a', 'b', 'c']
>>> 'a**b**c'.split('*')
['a', '', 'b', '', 'c']
>>> 'a\nb\nc'.split()
['a', 'b', 'c']
>>> 'a b c'.split()
['a', 'b', 'c']
>>> 'a b c'.split(',')
['a b c']
```

  - Three commonly used separators are `','`, **`'\n'`**, and `' '`

- If no separator is specified, the split method uses whitespace (newline, tab, or space) as the separator

- The **join** method turns a list of strings into a single string consisting of the elements of the list concatenated together and separated by a specified separator

```
line = ['To', 'be', 'or', 'not', 'to', 'be.']
string = ' '.join(line)
print(string)
krispies = ['Snap', 'Crackle', 'Pop']
print(', '.join(krispies))
```

[Run]

```
To be or not to be.
Snap, Crackle, Pop
```

Tuple

- Tuples, like lists, are ordered sequences of items but are immutable
  - Tuples have no **append**, **extend**, or **insert** methods
  - All other list functions and methods apply to tuples, and its items can also be accessed by indices
  - Useful if you want to create a sequence that cannot be modified, especially by mistake
- Tuples are written as comma-separated sequences enclosed in parentheses, or they can often be written without the parentheses
  - The following two statements create tuple **t** and assign it the same value

```
t = ('a', 'b', 'c')

t = 'a', 'b', 'c'
```

- **print** statement always displays tuples enclosed in parentheses

```
zoo = ('python', 'elephant', 'penguin')
print('Number of animals in the zoo is', len(zoo))

new_zoo = 'monkey', 'camel', zoo
print('All animals in new zoo are', new_zoo)
print('Number of cages in the new zoo is', len(new_zoo))
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is',
    len(new_zoo) - 1 + len(new_zoo[2]))
```

[Run]

```
Number of animals in the zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python',
'elephant', 'penguin'))
Number of cages in the new zoo is 3
Animals brought from old zoo are ('python', 'elephant',
'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

- A statement such as

```
(x, y, z) = (5, 6, 7)
```

creates three variables and assigns values to them

- Can also be written as

```
x, y, z = 5, 6, 7
```

which can be thought of as making three variable assignments with a single statement

```
x = 5
y = 6
x, y = y, x
print(x, y)
```

[Run]

```
6 5
```

Sequence

- Major features of sequences (i.e., strings, lists, and tuples):
  - Membership test
  - Indexing operation
  - Slicing operation: retrieves a part of the sequence

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'

# Indexing or 'Subscription' operation #
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])

# Slicing on a list #
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])

# Slicing on a string #
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

[Run]

```
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Character 0 is s
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
```

```
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

- The index $-k$ refers to the kth last item in the sequence
- The slice returned starts at the *start* position and ends just before the *end* position
- The numbers are optional in a slicing operation but the colon isn't
  - ◯ If the first number is not specified, Python will start at the beginning of the sequence
  - ◯ If the second number is left out, Python will stop at the end of the sequence

- You can also provide a third argument for the slice, which is the *step* for the slicing (by default, the step size is 1)

```
>>> shoplist = ['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::1]
['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::2]
['apple', 'carrot']
>>> shoplist[::3]
['apple', 'banana']
>>> shoplist[::-1]
['banana', 'carrot', 'mango', 'apple']
>>> shoplist[::-2]
['banana', 'mango']
>>> shoplist[::]
['apple', 'mango', 'carrot', 'banana']
```

- Lists of tuples play a prominent role in analyzing data
- If `L` is a list of tuples, then `L[0]` is the first tuple
  - `L[0][0]` is the first item of the first tuple

- `L[-1]` (same as `L[len(L) - 1]`) is the last tuple
- `L[-1][-1]` is the last item of the last tuple

```
regions = [('Northeast', 55.3), ('Midwest', 66.9),
           ('South', 114.6), ('West', 71.9)]
print('The 2010 population of the', regions[1][0], 'was',
      regions[1][1], 'million.')
totalPop = regions[0][1] + regions[1][1] + regions[2][1]
+ regions[3][1]
print('Total 2010 population of the U.S: {0:.1f} million.'
.format(totalPop))
```

[Run]

```
The 2010 population of the Midwest was 66.9 million.
Total 2010 population of the U.S: 236.8 million.
```

- The `list` function converts tuples or strings to lists

```
>>> list(('a', 'b'))
['a', 'b']
>>> list("Python")
['P', 'y', 't', 'h', 'o', 'n']
>>> "Python".split()
['Python']
>>> "P y t h o n".split(' ')
['P', 'y', 't', 'h', 'o', 'n']
>>> "P y t h o n".split()
['P', 'y', 't', 'h', 'o', 'n']
```

References

- When a mutable object is assigned to a variable, the variable name just points to the memory where the object is stored

```
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
# mylist is just another name pointing to the same object!
mylist = shoplist
# I purchased the first item, so I remove it from the list
del mylist[0]
```

```
print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that both shoplist and mylist print the same list
# without the 'apple' confirming that they point to the same
# object

print('Copy by making a full slice')
# Make a copy by using a full slice
mylist = shoplist[:]

# Remove first item
del mylist[0]
print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that now the two lists are different
```

[Run]

```
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

- Slicing operation should be used to make a copy of a sequence
  - If you just assign the variable name to another name, both of them will *refer* to the same object

```
>>> x = [1, 2, 3, 4]
>>> y = x                # y points to x
>>> z = x[:]             # z is a copy of x
>>> x.append(5)
>>> y
[1, 2, 3, 4, 5]
>>> z                    # z is still a copy of previous x
[1, 2, 3, 4]
>>> w = y.copy()         # another way of copying a list
>>> del y[0]
>>> x                    # x and y point to the same thing
[2, 3, 4, 5]
>>> y
[2, 3, 4, 5]
>>> z
[1, 2, 3, 4]
>>> w
[1, 2, 3, 4, 5]
```

- When a variable is created with an assignment statement, the value on the right side becomes an object in memory, and the variable references (that is, points to) that object
- When a list is altered after being assigned to a variable, changes are made to the object in the referenced memory location
- However, when a variable whose value is a number, string, or tuple, has its value changed, Python designates a new memory location to hold the new value and the variable references that new object
- We say that lists can be changed in place, but numbers, strings, and tuples cannot
- Objects that can be changed in place are called mutable, and objects that cannot be changed in place are called immutable

More about Strings

- The strings are objects of the class **`str`**
  - The next example shows more string methods that are useful
  - For a complete list of such methods, see **`help(str)`**

```
# This is a string object
name = 'Swaroop'

if name.startswith('Swa'):
    print('Yes, the string starts with "Swa"')

if 'a' in name:
    print('Yes, it contains the character "a"')

if name.find('war') != -1:
    print('Yes, it contains the string "war"')
```

```
delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

[Run]

```
Yes, the string starts with "Swa"
Yes, it contains the character "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

04 Functions

Function Definition

- Functions are commonly defined by statements of the form

  def functionName(*par1*, *par2*,...):

  indented block of statements

  return *expression*

  where ***par1***, ***par2*** , ... are the parameters and the ***expression*** evaluates to a literal of any type

- When defining a function (or a class), it is useful to document it using a string literal called the **doc** string at the start of the definition
  - The **doc** string is enclosed in three sets of double quotes **(""")**
  - It will be placed in the **__doc__** attribute of the definition
  - It can be accessed using **print(functionName.__doc__)**
- We can also use **help(object)** to get the description of **object**

```
def print_max(a, b): # this function has no return statement
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')

# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

```
[Run]
4 is maximum
7 is maximum
```

- The return statement is used to return from a function, i.e., break out of the function
  - We can optionally return a value from the function as well

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
print(maximum(5, 5))

[Run]
3
The numbers are equal
```

* There is a built-in function called **`max`** that implements the 'find maximum' functionality

- Every function implicitly contains a **`return None`** statement at the end unless you have written your own return statement
  - **`None`** is a special type in Python that represents nothingness
  - A **`return`** statement without a value is equivalent to **`return None`**

```
def f():
    return

print(f())

[Run]
None
```

Scope of Variables

- Local variable:
  - A variable created inside a function
  - Can only be accessed by statements inside that function
  - Local variables are recreated each time the function is called

    (They cease to exist when the function is exited)
  - Variables created in two different functions with the same name are treated as completely different variables

    (i.e., variable names are local to the function)
- Global variable:
  - A variable recognized everywhere in a program

```
x = 50  # This x is a global variable

def func(x):
    print('x is', x)  # This x is a local variable
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

[Run]

```
x is 50
Changed local x to 2
x is still 50
```

```
def main():
    ## Demonstrate the scope of variables.
    x = 2
    print(str(x) + ": function main")
    trivial()
    print(str(x) + ": function main")
```

```
def trivial():
    x = 3
    print(str(x) + ": function trivial")

main()
```

[Run]

```
2: function main
3: function trivial
2: function main
```

- One way to make a variable global is to place the assignment statement that creates it at the top of the program
- Any function can read the value of a global variable—however, the value cannot be altered inside a function unless the altering statement is preceded by a statement of the form

  global *globalVariableName*

```
x = 50

def main():
    func()
    print('Value of x is', x)

def func():
    global x  # Can declare more than one global variable

    print('x is', x)
    x = 2
    print('Changed global x to', x)

main()
```

[Run]

```
x is 50
Changed global x to 2
Value of x is 2
```

- Named constant:
  - A special constant that will be used several times in the program
  - Created as a global variable whose name is written in uppercase letters with words separated by underscore characters

```
INTEREST_RATE = 0.04
MINIMUM_VOTING_AGE = 18

 . . . . . .
interestEarned = INTEREST_RATE * amountDeposited
 . . . . . .
if (age >= MINIMUM_VOTING_AGE):
    print("You are eligible to vote.")
 . . . . . .
```

  - To change the value of a named constant at a later time, you need to alter just one line of code at the top of the program

More Examples

```
def main():
    ## Extract the first name from a full name.
    fullName = input("Enter a person's full name: ")
    print("First name:", firstName(fullName))

def firstName(fullName):
    firstSpace = fullName.index(" ")
    givenName = fullName[:firstSpace]
    return givenName

main()
```

[Run]

```
Enter a person's full name: Franklin Delano Roosevelt
First name: Franklin
```

  - The **`index`** method returns the index of the given character in the sequence

```
def main():
    ## Calculate a person's weekly pay.
    hourlyWage = float(input("Enter the hourly wage: "))
    hoursWorked = int(input("Enter # hours worked: "))
    earnings = pay(hourlyWage, hoursWorked)
    print("Earnings: ${0:,.2f}".format(earnings))

def pay(wage, hours):
    if hours <= 40:
        amount = wage * hours
    else:
        amount = (wage * 40) + ((1.5) * wage * (hours - 40))
    return amount

main()
```

[Run]

```
Enter the hourly wage: 24.50
Enter # hours worked: 45
Earnings: $1,163.75
```

```
def main():
    ## Display the vowels appearing in a word.
    word = input("Enter a word: ")
    listOfVowels = occurringVowels(word)
    print("The following vowels occur in the word:", end='')
    stringOfVowels = " ".join(listOfVowels)
    print(stringOfVowels)

def occurringVowels(word):
    word = word.upper()
    vowels = ('A', 'E', 'I', 'O', 'U')
    includedVowels = []
    for vowel in vowels:
        if (vowel in word) and (vowel not in includedVowels):
            includedVowels.append(vowel)
    return includedVowels

main()
```

[Run]

```
Enter a word: important
The following vowels occur in the word: A I O
```

- This is an example of a list-valued function

```
INTEREST_RATE = .04    # annual rate of interest

def main():
    ## Calculate the balance and interest earned
    (deposit, numberOfYears) = getInput()
    bal, intEarned = balAndInterest(deposit, numberOfYears)
    displayOutput(bal, intEarned)

def getInput():
    deposit = int(input("Enter the amount of deposit: "))
    numberOfYears = int(input("Enter # of years: "))
    return (deposit, numberOfYears)
def balAndInterest(principal, numYears):
    balance = principal * ((1 + INTEREST_RATE) ** numYears)
    interestEarned = balance - principal
    return (balance, interestEarned)
def displayOutput(bal, intEarned):
    print("Balance: ${0:,.2f}   Interest Earned: ${1:,.2f}"
          .format(bal, intEarned))

main()
```

[Run]

```
Enter the amount of deposit: 10000
Enter # of years: 10
Balance: $14,802.44   Interest Earned: $4,802.44
```

- The function **balAndInterest** does not return two values but just one value that is a tuple containing two values
- The fourth line of the function **main** can be written

  (bal, intEarned) = balAndInterest(deposit, numberOfYears)

```
def main():
    ## Custom sort a list of words.
    list1 = ["democratic", "sequoia", "equals", "brrr",
             "break", "two"]
    list1.sort(key=len)
    print("Sorted by length in ascending order:")
    print(list1, '\n')
    list1.sort(key=numberOfVowels, reverse=True)
    print("Sorted by number of vowels in descending order:")
    print(list1)
```

```
def numberOfVowels(word):
    vowels = ('a', 'e', 'i', 'o', 'u')
    total = 0
    for vowel in vowels:
        total += word.count(vowel)
    return total

main()
```

[Run]

```
Sorted by length in ascending order:
['two', 'brrr', 'break', 'equals', 'sequoia', 'democratic']

Sorted by number of vowels in descending order:
['sequoia', 'democratic', 'equals', 'break', 'two', 'brrr']
```

- To create a custom sort by any criteria we choose we add the optional argument **`key=keyValue`** to the sort method
  - ◯ ***`keyValue`*** is the name of a function
  - ◯ The function takes each item of the list as input and returns the value of the property we want to sort on
- The argument **`reverse=True`** can be added to sort in descending order

● While the **`sort`** method alters the order of the items in a list, the **`sorted`** function returns a new ordered copy of a list

```
>>> list1 = ['white', 'blue', 'red']
>>> list2 = sorted(list1)
>>> list2
['blue', 'red', 'white']
>>> sorted(list1, reverse=True)
['white', 'red', 'blue']
>>> sorted(list1, key=len)
['red', 'blue', 'white']
>>> list1
['white', 'blue', 'red']
>>> sorted('spam')
['a', 'm', 'p', 's']
```

- While the **sort** method only can be used with lists, the **sorted** function also can be used with lists, strings, and tuples

Library Modules

- A library module is a file with the extension **.py** containing functions and variables that can be used (we say imported) by any program
  - The library module can be created in IDLE or any text editor and looks like an ordinary Python program
- To gain access to the functions and variables of a library module, place a statement of the form **import *moduleName*** at the beginning of the program
  - Any function from the module can be used in the program by prepending the function name with the module name followed by a period
- Assuming that the function **pay** in the example program on p.51 is contained in a file named **finance.py** that is located in the same folder as the example, the example could be rewritten as

```
import finance

def main():
    ## Calculate a person's weekly pay.
    hourlyWage = float(input("Enter the hourly wage: "))
    hoursWorked = int(input("Enter # hours worked: "))
    earnings = finance.pay(hourlyWage, hoursWorked)
    print("Earnings: ${0:,.2f}".format(earnings))

main()
```

- The only changes in the program are the replacements of

◯ the definition of **pay** function with the import statement

◯ **pay** in the assignment statement with **finance.pay**

- There are different ways to import modules, as e.g., to import some functions from the **random** module:

(1) **from random import randint, choice**

(2) **from random import ***

(3) **import random**

(1) imports just two functions from the module

(2) imports every function from the module

  - You should usually avoid doing this, as the module may contain some names that will interfere with your own variable names (e.g., a variable **total** and a function **total**, in imported module)

(3) imports an entire module without interference

  - To use a function from the module, preface it with random followed by a dot, e.g., **random.randint(1,10)**

- The **as** keyword can be used to change the name that your program uses to refer to a module or things from a module:

```
import numpy as np
from tools import combinations_with_replacement as cwr
from math import log as ln
```

- Usually, these statements go at the beginning of the program, but they can go anywhere as long as they come before the code that uses the module

- Some modules from the Python standard library:

Module	Some Tasks Performed by Its Functions
`os`	Delete and rename files
`os.path`	Determine whether a file exists in a specified folder. This module is a submodule of os
`pickle`	Store objects (such as dictionaries, lists, and sets) in files and retrieve them from files
`random`	Randomly select numbers and subsets
`tkinter`	Enable programs to have a graphical user interface
`turtle`	Enable turtle graphics

List Comprehension

- If **`list1`** is a list, then the following statement creates a new list, **`list2`**, and places **`f(item)`** into the list for each **`item`** in **`list1`**

```
list2 = [f(x) for x in list1]
```

where **`f`** is either a Python built-in function or a user-defined function

```
>>> list1 = ['2', '5', '6', '7']
>>> [int(x) for x in list1]
[2, 5, 6, 7]
>>> def g(x):
        return(int(x) ** 2)

>>> [g(x) for x in list1]
[4, 25, 36, 49]
>>>
```

- The **`for`** clause in a list comprehension can optionally be followed by an **`if`** clause

```
>>> [g(x) for x in list1 if int(x) % 2 == 1]
[25, 49]
>>>
```

- List comprehension can be applied to objects other than lists, such as, strings, tuples, and arithmetic progressions generated by range functions

```
>>> [ord(x) for x in "abc"]
[97, 98, 99]
>>> [x ** .5 for x in (4, -1, 9) if x >= 0]
[2.0, 3.0]
>>> [x ** 2 for x in range(3)]
[0, 1, 4]
>>>
```

  - If **`str`** is any single-character string, then **`ord(str)`** is the ASCII value of the character
  - If **`n`** is a nonnegative number, then **`chr(n)`** is the single-character string consisting of the character with ASCII value **`n`**

```
>>> print(ord('A'))
65
>>> print(chr(65))
A
```

Default Argument Values

- Some (or all) of the parameters of a function can be made optional and have default values—values that are assigned to them when no values are passed to them
- A typical format for a function definition using default values is

```
def functionName(par1, par2, par3=value3, par4=value4):
```

- Caution: In a function definition, the parameters without default values must precede the parameters with default values

```
def main():
    say('Hello')
    say('World', 5)

def say(message, times=1):
    print(message * times)

main()
```

[Run]

```
Hello
WorldWorldWorldWorldWorld
```

Keyword Arguments

- Arguments can be passed to functions by using the names of the corresponding parameters instead of relying on position

```
def main():
    func(3, 7)
    func(25, c=24)
    func(c=50, a=100)
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

main()
```

[Run]

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

- Caution: Arguments passed by position must precede arguments passed by keyword

Lambda Expression

- Lambda expressions are one-line mini-functions
  - Compute a single expression
  - Cannot be used as a replacement for complex functions

lambda *par1, par2,...: expression*

```
names = ["Dennis Ritchie", "Alan Kay", "John Backus",
                "James Gosling"]
names.sort(key=lambda name: name.split()[-1])
nameString = ", ".join(names)
print(nameString)
```

[Run]

```
John Backus, James Gosling, Alan Kay, Dennis Ritchie
```

Top-Down Design

- Functions allow programmers to focus on the main flow of a complex task and defer the details of implementation
  - As a rule, a function should perform only one task, or several closely related tasks, and should be kept relatively small
  - Functions are used to break complex problems into small problems, to eliminate repetitive code, and to make a program easier to read by separating it into logical units
- The first function of a program is named main and sometimes will be preceded by import statements and global variables
  - All programs will end with the statement **`main()`** to call the program's main function

- The function main should be a supervisory function calling other functions according to the application’s logic

05 Data Processing and File Access

Reading Text Files

```
def main():
    ## Display the names of the first three presidents.
    file = "FirstPresidents.txt"
    displayWithForLoop(file)
    print()
    displayWithListComprehension(file)
    print()
    displayWithReadline(file)

def displayWithForLoop(file):
    infile = open(file, 'r')
    for line in infile:
        print(line.rstrip())
    infile.close()

def displayWithListComprehension(file):
    infile = open(file, 'r')
    listPres = [line.rstrip() for line in infile]
    infile.close()
    print(listPres)

def displayWithReadline(file):
    infile = open(file, 'r')
    line = infile.readline()
    while line != "":
        print(line.rstrip())
        line = infile.readline()
    infile.close()

main()
```

[Run]

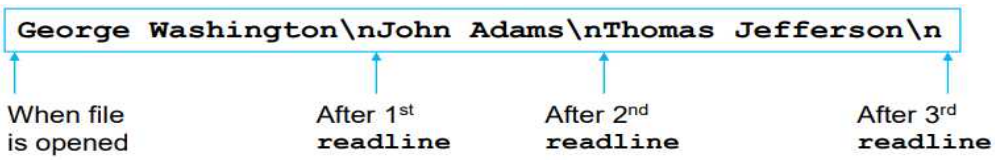
```
George Washington
John Adams
Thomas Jefferson

['George Washington', 'John Adams', 'Thomas Jefferson']

George Washington
John Adams
Thomas Jefferson
```

- When a text file is opened for input, a pointer is set to the beginning of the first line of the file
- Each time a statement of the form

  strVar = infile.readline()

  is executed, the current line is assigned to **strVar** and the pointer advances to the end of that line
- After all the lines of the file have been read, the **readline** method returns the empty string

FirstPresidents.txt



Creating Text Files

- A statement of the form

  outfile **= open(*fileName*, 'w')**

  creates and opens a new text file with the specified name for writing
- If ***list1*** is a list of strings, where each string ends with a newline character (**\n**), then the statement

  outfile**.writelines(*list1*)**

  writes each item of the list into the file as a line
- If the value of ***strVar*** is a string, the statement

  outfile**.write(*strVar*)**

  adds the value of ***strVar*** to the file

- After all writing has been finished, the file must be closed to guarantee that all data has been physically transferred to the disk

```
def main():
    ## Create two files containing the first three presidents.
    L = ["George Washington", "John Adams",
             "Thomas Jefferson"]
    outfile = open("FirstPresidents2.txt", 'w')
    createWithWrite(L, outfile)
    outfile = open("FirstPresidents3.txt", 'w')
    createWithWritelines(L, outfile)

def createWithWrite(L, outfile):
    for i in range(len(L)):
        outfile.write(L[i] + "\n")
    outfile.close()

def createWithWritelines(outfile):
# Append endline characters to the list's items.
    for i in range(len(L)):
        L[i] = L[i] + "\n"
    # Write the list's items to the file.
    outfile.writelines(L)
    outfile.close()

main()
```

[Run]

- If an existing file is opened for writing, its original content will be replaced by new content
- Each of the newly created files will look as follows when opened in a text editor

```
George Washington
John Adams
Thomas Jefferson
```

- The file **`States.txt`** contains the names of the U.S. states in the order they joined the union

- The following program uses this file to create a text file named **StatesAlpha.txt** containing the states in alphabetical order

```
def main():
    ## Create a text file containing the 50 states in
    #  alphabetical order.
    statesList = createListFromFile("States.txt")
    createSortedFile(statesList, "StatesAlpha.txt")

def createListFromFile(fileName):
    infile = open(fileName, 'r')
    desiredList = [line.rstrip() for line in infile]
    infile.close()
    return desiredList

def createSortedFile(listName, fileName):
    listName.sort()
    for i in range(len(listName)):
        listName[i] = listName[i] + "\n"
    outfile = open(fileName, 'w')
    outfile.writelines(listName)
    outfile.close()

main()
```

[Run]

- **StatesAlpha.txt** will look as follows when opened in a text editor

```
Alabama
Alaska
  ⋮
Wisconsin
Wyoming
```

- The file **USPres.txt** contains the U.S. presidents in the order they served and the file **VPres.txt** contains the names of the people who served as vice presidents of the U.S.
  - The following program creates a file named **Both.txt** containing the names of the presidents who also served as vice president

```
def main():
    ## Create a file of the presidents who also served as
    #  vice-presidents.
    vicePresList = createListFromFile("VPres.txt")
    createNewFile(vicePresList, "USPres.txt", "Both.txt")

def createListFromFile(fileName):
    infile = open(fileName, 'r')
    desiredList = [line.rstrip() for line in infile]
    infile.close()
    return desiredList

def createNewFile(listName, oldFileName, newFileName):
    infile = open(oldFileName, 'r')
    outfile = open(newFileName, 'w')
    for person in infile:
        if person.rstrip() in listName:
            outfile.write(person)
    infile.close()
    outfile.close()

main()
```

[Run]

- **`Both.txt`** will look as follows when opened in a text editor

```
John Adams
Thomas Jefferson
  ⋮
Gerald Ford
George H. W. Bush
```

Adding Lines to an Existing Text File

- A statement of the form

  `outfile` **`= open(`*** ***`fileName`*** **`, 'a')`**

  allows the program to add lines to the end of the specified file (the file is said to be opened for append)

```
def main():
    ## Add next three presidents to the file containing
    #  first three presidents.
    outfile = open("FirstPresidents.txt", 'a')
    list1 = ["James Madison\n", "James Monroe\n"]
    outfile.writelines(list1)
    outfile.write("John Q. Adams\n")
    outfile.close()

main()
```

[Run]

- The file **`FirstPresidents.txt`** will now look as follows when opened in a text editor

```
George Washington
John Adams
Thomas Jefferson
James Madison
James Monroe
John Q. Adams
```

Altering Items in a Text File

- Altering, inserting, or deleting a line of a text file cannot be made directly
  - A new file must be created by reading each item from the original file and recording it, with the changes, into the new file
  - The old file is then erased, and the new file is renamed with the name of the original file
- To gain access to the functions needed for these tasks, we must first import the standard library module os with the statement

import os

and then delete the specified file using the statement

os.remove(*fileName*)

- The statement

  os.rename(*oldFileName*, *newFileName*)

  will change the name and possibly the path of a file

- The remove and rename functions cannot be used with open files
  - The second argument of the rename function cannot be the name of an existing file
  - An error message is generated if the file to be removed, renamed, or opened for reading does not exist
- To verify if a file exists before attempting to rename, delete, or read it, we can use

  os.path.isfile(*fileName*)

  that returns **True** if the specified file exists and **False** otherwise
- Assume that the current folder does not contain a file named **ABC.txt** when the following program is run

```
import os.path

if os.path.isfile("ABC.txt"):
    print("File already exists.")
else:
    infile = open("ABC.txt", 'w')
    infile.write("a\nb\nc\n")
    infile.close()
```

  - What happens the first time the program is run?

- What happens the second time the program is run?

Sets

- A set is an unordered collection of items (referred to as elements) with no duplicates
  - Sets can contain numbers, strings, tuples, and Boolean values
  - Some examples of sets are

    {'spam', 'ni'}, {3, 4, 7},

    {True, 'eleven', 7}, {'a', 'b', (3, 4)}
  - Sets cannot contain lists or other sets
  - Since the elements have no order, they cannot be indexed
    - ◯ Slicing and list methods such as **sort** and **reverse** are meaningless
- Useful set operations include membership test, subset test, set intersection, and so on

```
>>> bri = {'brazil', 'russia', 'india'}
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bri.remove('russia')  # or bri.discard('russia')
>>> bri
{'brazil', 'india'}
>>> bric
{'brazil', 'russia', 'india', 'china'}
>>> bric.issuperset(bri)  # or bri.issubset(bric)
True
>>> bri & bric  # or bri.intersection(bric)
{'brazil', 'india'}
```

```
>>> bric.difference(bri)  # or bric - bri
{'russia', 'china'}
>>> bric.union(bri)  # or bric | bri
{'brazil', 'india', 'china', 'russia'}
```

- List, tuple, and set can be converted to one another by using the functions **`list`**, **`tuple`**, and **`set`**

```
>>> words = ['nudge', 'nudge', 'wink', 'wink']
>>> tuple(words)
('nudge', 'nudge', 'wink', 'wink')
>>> terms = set(words)
>>> print(terms)
{'wink', 'nudge'}
>>> list(terms)
['wink', 'nudge']
>>> tuple(terms)
('wink', 'nudge')
>>> alpha =('a', 'b', 'c')
>>> set(alpha)
{'a', 'c', 'b'}
>>> terms.clear()  # clear is a set method
>>> terms
set()
```

- Although the elements of a set cannot be ordered, they can be placed into a list in a customized order using the **`sorted`** function

```
>>> bric = {'brazil', 'china', 'india', 'russia'}
>>> bric
{'china', 'india', 'brazil', 'russia'}
>>> sorted(bric)
['brazil', 'china', 'india', 'russia']
>>> sorted(bric, key=len, reverse=True)
['brazil', 'russia', 'china', 'india']
>>> bric
{'china', 'india', 'brazil', 'russia'}
```

- Like lists, sets can be created with comprehension

```
>>> {x * x for x in range(-3, 3)}
{0, 9, 4, 1}
```

- The following is a rewrite of the example program on p.65 using set methods to create a file containing the names of presidents who also served as vice president

```
def main():
    ## Create a file of the presidents who also served
    #  as vice-presidents.
    vicePresSet = createSetFromFile("VPres.txt")
    presSet = createSetFromFile("USPres.txt")
    bothPresAndVPresSet = createIntersection(vicePresSet,
                                             presSet)
    writeNamesToFile(bothPresAndVPresSet, "PresAndVPres.txt")

def createSetFromFile(fileName):
    # Assume that the last line of the file ends with
    # a newline character.
    infile = open(fileName, 'r')
    namesSet = {name for name in infile}
    infile.close()
    return namesSet

def createIntersection(set1, set2):
    return set1.intersection(set2)

def writeNamesToFile(setName, fileName):
    outfile = open(fileName, 'w')
    outfile.writelines(setName)
    outfile.close()

main()
```

- Set operations (**`words = {'spam', 'ni'}`**)

Methods and Functions	Example	Value of Set	Description
add	words.add("eggs")	{"spam", "ni", "eggs"}	adds item to set
discard	words.discard{"ni"}	{"spam"}	removes specified item
clear	words.clear()	{}	{} is the empty set
set	set([3, 7, 3])	{3, 7}	convert a list to a set
	set((3, 7, 3))	{3, 7}	convert a tuple to a set

```
>>> {1, 2, 3} | {3, 4}      # set union
{1, 2, 3, 4}
>>> {1, 2, 3} & {3, 4}      # set intersection
{3}
>>> {1, 2, 3} - {3, 4}      # set difference
{1, 2}
>>> {1, 2, 3} ^ {3, 4}      # symmetric difference
{1, 2, 4}
>>> 3 in {1, 2, 3}          # is an element of
True
>>> x = {1, 2, 3} - {3, 4} # x = {1, 2}
>>> x.add(5)                # x = {1, 2, 5}
>>> y = x.copy()            # y is a copy of x
>>> y.discard(1)
>>> y
{2, 5}
>>> x
{1, 2, 5}
```

CSV Files

- Text files considered so far had a single piece of data per line
- Consider CSV (comma separated values) formatted file
  - Several items of data on each line
  - Items separated by commas
- The file **`UN.txt`** contains the members of UN
  - Countries listed in alphabetical order
  - Each record contains data about a country: name, continent, population (in million), land area (in square miles)

```
Canada,North America,34.8,3855000
France,Europe,66.3,211209
New Zealand,Australia/Oceania,4.4,103738
Nigeria,Africa,177.2,356669
Pakistan,Asia,196.2,310430
Peru,South America,30.1,496226
```

Accessing the Data in a CSV File

- The **`split`** method is used to access the fields

```
def main():
    ## Display the countries in a specified continent.
    continent = input("Enter the name of a continent: ")
    continent = continent.title()   # Allow for all lower
    if continent != "Antarctica":   # case letters.
        infile = open("UN.txt", 'r')
        for line in infile:
            data = line.split(',')
            if data[1] == continent:
                print(data[0])
    else:
        print("There are no countries in Antarctica.")

main()
```

[Run]

```
Enter the name of a continent: South America
Argentina
Bolivia
Brazil
Chile
Colombia
Ecuador
Guyana
Paraguay
Peru
Suriname
Uruguay
Venezuela
```

Analyzing the Data in a CSV File with a List

- Data can be analyzed by placing data into a list
  - Items of the list are other lists holding the contents of a single line of the file

```
def main():
    ## Create a file containing all countries and areas,
    ## ordered by area.
    ## Display first five lines of the file.
```

```
    countries = placeRecordsIntoList("UN.txt")
    countries.sort(key=lambda country: country[3],
                   reverse=True)  #sort by area
    displayFiveLargestCountries(countries)
    createNewFile(countries)  # Create file of countries and
                              # their areas.

def placeRecordsIntoList(fileName):
    infile = open(fileName, 'r')
    listOfRecords = [line.rstrip() for line in infile]
    infile.close()
    for i in range(len(listOfRecords)):
        listOfRecords[i] = listOfRecords[i].split(',')
        listOfRecords[i][2] = eval(listOfRecords[i][2])
                                                  # population
        listOfRecords[i][3] = eval(listOfRecords[i][3])
                                                  # area
    return listOfRecords

def displayFiveLargestCountries(countries):
    print("{0:20}{1:9}".format("Country", "Area (sq. mi.)"))
    for i in range(5):
        print("{0:20}{1:9,d}".format(countries[i][0],
                                     countries[i][3]))

def createNewFile(countries):
     outfile = open("UNbyArea.txt", 'w')
     for country in countries:
         outfile.write(country[0] + ',' + str(country[3])
                       + "\n")
     outfile.close()

main()
```

[Run]

```
Country             Area (sq. mi.)
Russian Federation  6,592,800
Canada              3,855,000
United States       3,794,066
China               3,696,100
Brazil              3,287,597
```

- The first three lines of the CSV file **`UNbyArea.txt`** are

```
Russian Federation,6592800
Canada,3855000
United States,3794066
```

- CSV files can be converted to Excel spreadsheets
  - Open UN.txt file in Excel, select comma when asked for delimiter
- Spreadsheets can be converted to CSV files
  - Click on “Save As” from the FILE menu, choose “CSV (Comma delimited)” in the “Save as type” dropdown box

Dictionary

- A dictionary is a collection of comma-separated pairs of the form

  `d = {key1:value1, key2:value2, . . .}`

- The keys must be unique immutable objects (such as strings, numbers, or tuples)
- The value associated with **`key1`** is given by the expression **`d[key1]`**
- The **`dict`** function converts a list of two-item lists or two-item tuples into a dictionary

```
>>> list1 = [["one", 1], ["two", 2], ["three", 3]]
>>> dict(list1)
{'one': 1, 'two': 2, 'three': 3}
>>> list2 = [("one", 1), ("two", 2), ("three", 3)]
>>> dict(list2)
{'one': 1, 'two': 2, 'three': 3}
```

```
addr = { 'Swaroop'   : 'swaroop@swaroopch.com',
         'Larry'     : 'larry@wall.org',
         'Matsumoto' : 'matz@ruby-lang.org',
         'Spammer'   : 'spammer@hotmail.com'
       }
print("Swaroop's address is", addr['Swaroop'])
```

```
# Deleting a key-value pair
del addr['Spammer']
print('\nThere are {} contacts in the address-book\n' \
      .format(len(addr)))

for name, address in list(addr.items()):
    print('Contact {} at {}'.format(name, address))

# Adding a key-value pair
addr['Guido'] = 'guido@python.org'
if 'Guido' in addr:
    print("\nGuido's address is", addr['Guido'])
```

[Run]

```
Swaroop's address is swaroop@swaroopch.com

There are 3 contacts in the address-book

Contact Swaroop at swaroop@swaroopch.com
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org

Guido's address is guido@python.org
```

- The operation `list(d.items())` returns a list of tuples where each tuple contains a key value pair
- New key-value pairs can be added by simply using the indexing operator to access a key and assign its value

- Dictionary operations:

Operation	Description
len(d)	number of items (that is *key:value* pairs) in the dictionary
x in d	has value True if *x* is a key of the dictionary
x:*y* in d	has value True if *x*:*y* is an item of the dictionary. Otherwise, has value False.
x:*y* not in d	has value True if *x*:*y* is not an item of the dictionary. Otherwise, has value False.
d[*key1*] = *value1*	if *key1* is already a key in the dictionary, changes the value associated with *key1* to *value1*; otherwise, adds the item *key1*:*value1* to the dictionary.

d[*key1*]	returns the value associated with *key1*. Raises an error if *key1* is not a key of d.
d.get(*key1*, *default*)	if *key1* is not a key of the dictionary, returns the default value. Otherwise, returns the value associated with *key1*.
list(d.keys())	returns a list of the keys in the dictionary.
list(d.values())	returns a list of the values in the dictionary.
list(d.items())	returns a list of two-tuples of the form (*key*, *value*) where d(*key*) = *value*.
list(d)	returns a list of the keys in the dictionary.
tuple(d)	returns a tuple of the keys in the dictionary.
set(d)	returns a set of the keys in the dictionary.
c = {}	creates an empty dictionary.
c = dict(d)	creates a copy of the dictionary d.
del d[*key1*]	removes the item having *key1* as key Raises an exception if *key1* is not found
d.clear()	removes all items (that is *key:value* pairs) from the dictionary.
for k in d:	iterates over all the keys in the dictionary.
d.update(c)	merges all of dictionary c's entries into dictionary d. If two items have the same key, the value from c replaces the value from d.
max(d)	largest value of d.keys(), provided all keys have the same data type.
min(d)	smallest value of d.keys(), provided all keys have the same data type.

Using a Dictionary to Simplify a Long if-elif

```
def main():
    ## Determine an admission fee based on age group.
    print("Enter the person's age group ", end="")
    ageGroup = input("(child, minor, adult, or senior): ")
    print("The admission fee is",
          determineAdmissionFee(ageGroup), "dollars." )

def determineAdmissionFee(ageGroup):
    if ageGroup == "child":      # age < 6
        return 0                 # free
    elif ageGroup == "minor":    # age 6 to 17
        return 5                 # $5
    elif ageGroup == "adult":    # age 18 to 64
        return 10
    elif ageGroup == "senior":   # age >= 65
        return 8

main()
```

- The rewrite of **`determineAdmissionFee`** function below, replaces the **`if-elif`** statement with a discionary

```
def main():
    ## Determine an admission fee based on age group.
    print("Enter the person's age group ", end="")
    ageGroup = input("(child, minor, adult, or senior): ")
    print("The admission fee is",
          determineAdmissionFee(ageGroup), "dollars." )

def determineAdmissionFee(ageGroup):
    dict = {"child":0, "minor":5, "adult":10, "senior":8}
    return dict[ageGroup]

main()
```

[Run]

```
Enter the person's age group (child, minor, adult, or senior):
adult
The admission fee is 10 dollars.
```

Using a Dictionary as a Frequency Table

```
def main():
    ## Analyze word frequencies in the Gettysburg Address,
    ## which is written in a single line.
    listOfWords = formListOfWords("Gettysburg.txt")
    freq = createFrequencyDictionary(listOfWords)
    displayWordCount(listOfWords, freq)
    displayMostCommonWords(freq)

def formListOfWords(fileName):
    infile = open(fileName)
    originalLine = infile.readline().lower()
    # Remove punctuation marks from the line.
    line = ""
    for ch in originalLine:
        if ('a' <= ch <= 'z') or (ch == " "):
            line += ch
    # Place the individual words into a list.
    listOfWords = line.split()
    return listOfWords

def createFrequencyDictionary(listOfWords):
    ## Create dictionary with each item having the form
    ## word:word frequency.
    freq = {}   # an empty dictionary
    for word in listOfWords:
        freq[word] = 0
```

```
    for word in listOfWords:
        freq[word] = freq[word] + 1
    return freq

def displayWordCount(listOfWords, freq):
    print("The Gettysburg Address contains", len(listOfWords),
          "words.")
    print("The Gettysburg Address contains", len(freq),
          "different words.")
    print()

def displayMostCommonWords(freq):
    ## Common words are those with frequency > 5.
    print("The most common words and their frequencies are:")
    listOfMostCommonWords = []
    for word in freq.keys():
        if freq[word] >= 6:
            listOfMostCommonWords.append((word, freq[word]))
    listOfMostCommonWords.sort(key=lambda x: x[1],
                               reverse=True)
    for item in listOfMostCommonWords:
        print("   ", item[0] + ':', item[1])

main()
```

[Run]

```
The Gettysburg Address contains 268 words.
The Gettysburg Address contains 139 different words.

The most common words and their frequencies are:
    that: 13
    the: 11
    we: 10
    to: 8
    here: 8
    a: 7
    and: 6
```

Storing Dictionaries in Binary Files

- Methods that store dictionaries to, and retrieve dictionaries from binary files must be imported from a module named pickle
  - Binary format can only be accessed by special readers

```
import pickle

outfile = open(filename,'wb')
pickle.dump(dictionaryName, outfile)
outfile.close()

infile = open(filename, 'rb')
dictionaryName = pickle.load(infile)
infile.close()
```

- The extension for binary files is “dat” (e.g., **"UNdict.dat"**)

Dictionary-Valued Dictionaries

- Dictionary’s values can be any type of object including a dictionary
- Consider the dictionary

```
nations = {"Canada":{"cont":"North America",
                     "popl":34.8, "area":3855000},
           "France":{"cont":"Europe", "popl":66.3,
                     "area":211209},
           ...}
```

- The value of **nations["Canada"]** would be the dictionary

```
{"cont":"Europe", "popl":66.3, "area":211209}
```

- The value of nations**["France"]["cont"]** would be **Europe**

```
import pickle

def main():
    ## Display countries (and their population) from
    ## a specified continent.
    nations = getDictionary("UNdict.dat")
```

```
    print("Enter the name of a continent", end='')
    continent = input("other than Antarctica: ")
    continentDict = constructContinentNations(nations,
                                              continent)
    displaySortedResults(continentDict)

def getDictionary(fileName):
    infile = open(fileName, 'rb')
    countries = pickle.load(infile)
    infile.close()
    return countries

def constructContinentNations(nations, continent):
    ## Reduce the full 193 item dictionary to a dictionary
    ## consisting solely of the countries in the specified
    ## continent.
    continentDict = {}  # an empty dictionary
    for nation in nations:  # or nations.keys()
        if nations[nation]["cont"] == continent:
            continentDict[nation] = nations[nation]
    return continentDict

def displaySortedResults(dictionaryName):
    ## Display countries in descending order by population.
    continentList = sorted(dictionaryName.items(),
                           key=lambda k: k[1]["popl"], reverse=True)
    for k in continentList:
         print("  {0:s}: {1:,.2f}".format(k[0], k[1]["popl"]))

main()
```

[Run]

```
Enter the name of a continent other than Antarctica: Europe
  Russian Federation: 142.50
  Germany: 81.00
  United Kingdom: 66.70
  France: 66.30
  Italy: 61.70
  Spain: 47.70
  . . .
```

Using a Dictionary with Tuples as Keys

- **`USpresStatesDict.dat`** holds a dictionary whose items look like

```
('Kennedy', 'John'): 'Massachusetts'

('Reagan', 'Ronald'): 'California'
```

```
import pickle

def main():
    ## Displays the presidents from the given state ordered
    ## alphabetically by their last names.
    presDict = \
      createDictFromBinaryFile("USpresStatesDict.dat")
    state = getState(presDict)
    displayOutput(state, presDict)

def createDictFromBinaryFile(fileName):
    infile = open(fileName, 'rb')
    dictionary = pickle.load(infile)
    infile.close()
    return dictionary

def getState(dictName):
    state = input("Enter the name of a state: ")
    if state in dictName.values():
        return state
    else:
        return "There are no presidents from " + state + '.'

def displayOutput(state, dictName):
    if state.startswith("There"):
        print(state)
    else:
        print("Presidents from", state + ':')
        for pres in sorted(dictName):# in sorted list of names
            if dictName[pres] == state:
                print("   " + pres[1] + " " + pres[0])

main()
```

[Run]

```
Enter the name of a state: Virginia
Presidents from Virginia:
  Thomas Jefferson
  James Madison
  James Monroe
  John Tyler
  George Washington
```

Dictionary Comprehension

- Dictionaries can be created with dictionary comprehension, e.g.,

```
{x: x * x for x in range(4)}
```

- Dictionary comprehension can be used to extract a subset of a dictionary, e.g.,

```
NE = ["Maine", "Connecticut", "New Hampshire",
      "Massachusetts", "Vermont", "Rhode Island"]
subSet = {key: presDict{key] for key in presDict
                                if presDict[key] in NE}
```

06 Miscellaneous Topics

Exception Handling

- Exceptions occur due to circumstances beyond programmer's control
  - E.g., when invalid data are input or file cannot be accessed
- Even though the user is at fault
  - Programmer must anticipate exceptions
  - Include code to work around the occurrence

```
numDependents = int(input("Enter number of dependents: "))
taxCredit = 1000 * numDependents
print("Tax credit:", taxCredit)

[Run: When the user enters the word TWO]

Enter number of dependents:
ValueError: invalid literal for int() with base 10: ''
```

- Some common exceptions:

Exception Name	Description and Example
AttributeError	An unavailable functionality (usually a method) is requested for an object. `(2, 3, 1).sort()` or `print(x.endswith(3))` `# where x = 23`
FileNotFoundError	Requested file doesn't exist or is not located where expected. `open("NonexistantFile.txt", 'r')`
ImportError	Import statement fails to find requested module. `import nonexistentModule`
IndexError	An index is out of range. `letter = "abcd"[7]`
KeyError	No such key in dictionary. `word = d['c'])` `# where d = {'a':"alpha", 'b':"bravo"}`
NameError	The value of a variable cannot be found. `term = word` `# where word was never created`
TypeError	Function or operator receives the wrong type of argument. `x = len(23)` or `x = 6 / '2'` or `x = 9 + 'W'` or `x = abs(-3,4)`
ValueError	Function or operator receives right type of argument, but inappropriate value. `x = int('a')` or `L.remove(item)` `# where item not in list`
ZeroDivisionError	The second number in a division or modulus operation is 0. `num = 1 / 0` or `num = 23 % 0`

The *try* Statement

- The previous exception can be handled more robustly by protecting he code with a **`try`** statement

```
try:
    numDependents = int(input("Enter number of dependents: "))
except ValueError:
    print("\nYou did not respond with an integer value.")
    print("We will assume your answer is zero.")
    numDependents = 0
taxCredit = 1000 * numDependents
print("Tax credit:", taxCredit)
```

[Run: When the user enters the word TWO]

```
Enter number of dependents:

You did not respond with an integer value.
We will assume your answer is zero.
Tax credit: 0
```

- Three types of except clauses:

`except:`	(Its block is executed when any exception occurs.)
`except` ***`ExceptionType`*****`:`**	(Its block is executed only when the specified type of exception occurs.)
`except` ***`ExceptionType as exp`*****`:`**	(Its block is executed only when the specified type of exception occurs. Additional information about the problem is assigned to *exp*.)

- Program with different assumptions on exceptions:

```
def main():
    ## Display the reciprocal of a number in a file.
    try:
        fileName = input("Enter the name of a file: ")
        infile = open(fileName, 'r')
        num = float(infile.readline())
        print(1 / num)
    except FileNotFoundError as exc1:
        print(exc1)
```

```
    except TypeError as exc2:
        print(exc2)

main()
```

[Run: When `Numbers.txt` does not exit]

```
Enter the name of a file: Numbers.txt
[Errno 2] No such file or directory: 'Numbers.txt'
```

[Run: When the first line of `Numbers.txt` contains the word TWO]

```
Enter the name of a file: Numbers.txt
ValueError: could not convert string to float: 'TWO\n'
```

[Run: When the first line of `Numbers.txt` contains the number 2]

```
Enter the name of a file: Numbers.txt
0.5
```

- Program that uses exception handling to guarantee a proper response from the user:

```
def main():
    ## Request that the user enter a proper response.
    phoneticAlpha = {'a':"alpha", 'b':"bravo",'c':"charlie"}
    while True:
        try:
            letter = input("Enter a, b, or c: ")
            print(phoneticAlpha[letter])
            break
        except KeyError:
            print("Unacceptable letter was entered.")
main()
```

[Run]

```
Enter a, b, or c: d
Unacceptable letter was entered.
Enter a, b, or c: b
bravo
```

The *else* and *finally* Statement

- The following program uses exception handling to cope with the possibilities that the file is not found, the file contains a line that is not a number, or the file is empty

```
def main():
    ## Calculate the average and total of the numbers
    ## in a file.
    total = 0
    count = 0
    foundFlag = True
    try:
        infile = open("Numbers.txt", 'r')
    except FileNotFoundError:
        print("File not found.")
        foundFlag = False
    if foundFlag:
        try:
            for line in infile:
                count += 1
                total += float(line)
            print("average:", total / count)
        except ValueError:
            print("Line", count,
                  "could not be converted to a float")
            if count > 1:
                print("Average so far:", total / (count - 1))
                print("Total so far:", total)
            else:
                print("No average can be calculated.")
        except ZeroDivisionError:
            print("File was empty.")
        else:
            print("Total:", total)
        finally:
            infile.close()
```

- **`try`** statement can also include a single **`else`** clause
  - Follows the except clauses
  - Executed when no exceptions occur

- **`try`** statement can end with a **`finally`** clause
  - Usually used to clean up resources such as files that were left open
- **`try`** statement must contain either an **`except`** clause or a **`finally`** clause

Selecting Random Values

- The random module contains functions that randomly select items from a list and randomly reorder the items in a list

```
import random

elements = ["earth", "air", "fire", "water"]
print(random.choice(elements))
print(random.sample(elements, 2))
random.shuffle(elements)
print(elements)
print(random.randint(1, 5))  # random integer from 1 to 5
```

[Run]

```
earth
['air', 'earth']
['fire', 'water', 'air', 'earth']
5
```

```
import random
import pickle

infile = open("DeckOfCardsList.dat", 'rb')
deckOfCards = pickle.load(infile)
infile.close()
print(deckOfCards)
print()
pokerHand = random.sample(deckOfCards, 5)
print(pokerHand)
```

[Run]

```
['2♠', '3♠', '4♠', '5♠', '6♠', '7♠', '8♠', '9♠', '10♠',
'J♠', 'K♠', 'Q♠', 'A♠', '2♥', '3♥', '4♥', '5♥', '6♥', '7♥',
'8♥', '9♥', '10♥', 'J♥', 'K♥', 'Q♥', 'A♥', '2♣', '3♣',
'4♣', '5♣', '6♣', '7♣', '8♣', '9♣', '10♣', 'J♣', 'K♣',
```

```
'Q♣', 'A♣', '2◆', '3◆', '4◆', '5◆', '6◆', '7◆', '8◆', '9◆',
'10◆', 'J◆', 'K◆', 'Q◆', 'A◆']

['8♣', 'K♥', '10◆', 'Q♣', 'K♠']
```

```
## Items are selected from a list of six items.
## Selection probabilities are made different using
## if-elif-else statement.

import random

def main():
    for i in range(3):
        outcome = spinWheel()
        print(outcome, end="  ")

def spinWheel():
    n = random.randint(1, 20)
    if n > 15:
        return "Cherries"
    elif n > 10:
        return "Orange"
    elif n > 5:
        return "Plum"
    elif n > 2:
        return "Melon"
    elif n > 1:
        return "Bell"
    else:
        return "Bar"

main()
```

[Run]

```
Melon  Cherries  Orange
```

- A float value can be chosen randomly from an interval by using

  `random.uniform(a, b)`

- where **`a`** and **`b`** are the lower bound and upper bound, respectively

Swapping

- In most programming languages, we need to introduce a third variable when swapping the values of two variables, **`x`** and **`y`**:

```
temp = x

x = y

y = temp
```

- Python, however, provides a nice shortcut:

```
x, y = y, x
```

- Comma indicates that a tuple should be constructed
- All the expressions to the right of the assignment operator are evaluated before any of the assignments are made

Shortcuts

- Assignment shortcuts:

```
a = 0
b = 0
c = c
```

⇔

```
a = b = c = 0
```

- When L is a list with three elements in it:

```
x = L[0]
y = L[1]
z = L[2]
```

⇔

```
x, y, z = L
```

- We can assign three variables at a time:

```
x, y, z = 1, 2, 3
```

- We can swap variables like below:

```
x, y, z = y, z, x
```

- Shortcuts with condition:

```
if a == 0 and b == 0 and c == 0:
```

⇔

```
if a == b == c == 0:
```

```
if 1 < a and a < b and b < 5:
```

⇔

```
if 1 < a < b < 5:
```

Recursion

- A recursive solution to a problem has the general form:

```
If a base case is reached
   Solve the base case directly
else
   Repeatedly reduce the problem to a version increasingly
   close to a base case until it becomes a base case
```

```
def power(r, n):
    if n == 1:
        return r
    else:
        return r * power(r, n - 1)

print(power(2, 3))
```

[Run]

```
8
```

- A word is a palindrome if it reads the same forward and backward (e.g., racecar, kayak, pullup)

```
def isPalindrome(word):
    # Convert all letters to lowercase.
    word = word.lower()
    # Words of zero or one letters are palindromes.
    if len(word) <= 1:
        return True
    elif word[0] == word[-1]:  # First and last letters match.
        # Remove first and last letters.
        word = word[1:-1]
        return isPalindrome(word)
    else:
        return False
```

07 Object-Oriented Programming

Classes and Objects

- Data hiding is an important principle underlying object-oriented programming:
  - As much implementation detail as possible is hidden
- Object consists of two things:
  - Encapsulated data
    - ◯ Unauthorized access to some of an object's components is prevented
  - Methods that act on the data
    - ◯ Used to retrieve and modify the values within the object
- Programmer using an object is concerned only with
  - Tasks that the object can perform
  - Parameters used by these tasks (i.e., methods)

User-Defined Classes

- A class is a template from which objects are created
  - Specifies the properties and methods that will be common to all objects that are instances of that class
  - The data types **`str`**, **`int`**, **`float`**, **`list`**, **`tuple`**, **`dictionary`**, and **`set`**, are built-in Python classes
- Python allows users to create their own classes (i.e., data types)
  - Each class defined will have a specified set of methods

- Each object (instance) of the class will have its own value(s)

- Class definitions have the general form:

```
class ClassName:

    indented list of methods for the class
```

- Methods have **`self`** as their first parameter
  - When an object is created, each method's **`self`** parameter references the object
  - The **`__init__`** method (aka constructor) is automatically called when an object is created, assigning values to the instance variables (also called properties of the class)

```
class Rectangle:
    def __init__(self, width=1, height=1):
        self._width = width
        self._height = height

    def setWidth(self, width):
        self._width = width

    def setHeight(self, height):
        self._height = height

    def getWidth(self):
        return self._width

    def getHeight(self):
        return self._height

    def area(self):
        return self._width * self._height

    def perimeter(self):
        return 2 * (self._width + self._height)

    def __str__(self):
        return ("Width: " + str(self._width)
         + "\nHeight: " + str(self._height))
```

- The **__str__** method provides a customized way to represent the state (values of the instance variables) of an object as a string
- Classes can be typed directly into programs or stored in modules and brought into programs with an import statement
- An object, which is an instance of a class, is created with a statement of the form

```
objectName = ClassName(arg1, arg2, . . . )
```

or

```
objectName = modulName.ClassName(arg1, arg2, . . . )
```

```
import rectangle

# Create a rectangle of width 4 and height 5
r = rectangle.Rectangle(4, 5)
print(r)  # Uses the __str__ method to report the state
# Create a rectangle with the default values for width and
# height
r = rectangle.Rectangle()
print(r)
# Create a rectangle of width 4 and default height 1
r = rectangle.Rectangle(4)
print(r)
```

[Run]

```
Width: 4
Height: 5
Width: 1
Height: 1
Width: 4
Height: 1
```

```
import rectangle

r = rectangle.Rectangle()
# Use the mutator methods.
r.setWidth(4)
r.setHeight(5)
print("The rectangle has the following measurements:")
```

```
# Use the accessor methods.
print("Width is", r.getWidth())
print("Height is", r.getHeight())
# Use other methods.
print("Area is", r.area())
print("Perimeter is", r.perimeter())
```

[Run]

```
The rectangle has the following measurements:
Width is 4
Height is 5
Area is 20
Perimeter is 18
```

- Note:
  - **`r.setWidth(4)`** and **`r.setHeight(5)`** can be replaced by **`r._width = 4`** and **`r._height = 5`**, respectively
  - **`print("Width is", r.getWidth())`** and **`print("Height is", r.getHeight())`** can be replaced by **`print("Width is", r._width)`** and **`print("Height is", r._height)`**, respectively
- However, such replacement is considered poor programming style
- Instance variable names start with a single underscore so that they cannot be directly accessed from outside of the class definition
- Object-oriented programming hides the implementation of methods from the users of the class

Other Forms of the Initializer Method

- There are three other ways the initializer can be defined:

```
def __init__(self):
```

```
        self._width = 1
        self._height = 1

    def __init__(self, width=1):
        self._width = width
        self._height = 1

    def __init__(self, width, height):
        self._width = width
        self._height = height
```

- With the third form, the constructor statement creating an instance must provide two arguments

Other Methods in a Class Definition

```
def main():
    ## Calculate and display a student's semester letter
    ## grade.
    name = input("Enter student's name: ")
    midterm = float(input("Enter grade on midterm exam: "))
    final = float(input("Enter grade on final exam: "))
    # Create an instance of an LGstudent object.
    st = LGstudent(name, midterm, final)
    print("\nNAME\tGRADE")
    # Display student's name and semester letter grade.
    print(st)

class LGstudent:
    def __init__(self, name="", midterm=0, final=0):
        self._name = name
        self._midterm = midterm
        self._final = final

    def setName(self, name):
        self._name = name

    def setMidterm(self, midterm):
         self._midterm = midterm
```

```
    def setFinal(self, final):
         self._final = final

    def calcSemGrade(self):
        grade = (self._midterm + self._final) / 2
        grade = round(grade)
        if grade >= 90:
            return "A"
        elif grade >= 80:
            return "B"
        elif grade >= 70:
            return "C"
        elif grade >= 60:
            return "D"
        else:
            return "F"

    def __str__(self):
        return self._name + "\t" + self.calcSemGrade()

main()
```

[Run]

```
Enter student's name: Fred
Enter grade on midterm exam: 87
Enter grade on final exam: 92

NAME    GRADE
Fred    A
```

Lists of Objects

- Items of a list can be any data type including a user-defined class
- The following program uses a list where each item is an **`LGstudent`** object

```
import lgStudent

def main():
    ## Calculate and display students' semester letter grades.
    listOfStudents = []  # To holds objects each for a student
    carryOn = 'Y'
```

```
    while carryOn == 'Y':  # Repeat until user says 'N'
        st = lgStudent.LGstudent()
        # Obtain student's name and grades.
        name = input("Enter student's name: ")
        midterm = float(input
                ("Enter student's grade on midterm exam: "))
        final = float(input
                ("Enter student's grade on final exam: "))
        # Create an instance of an LGstudent object.
        st = lgStudent.LGstudent(name, midterm, final)
        listOfStudents.append(st)    # Insert object into list.
        carryOn = input("Do you want to continue (Y/N)? ")
        carryOn = carryOn.upper()
    print("\nNAME\tGRADE")
    # Display students, names and semester letter grades.
    for pupil in listOfStudents:
        print(pupil)

main()
```

[Run]

```
Enter student's name: Alice
Enter student's grade on midterm exam: 88
Enter student's grade on final exam: 94
Do you want to continue (Y/N)? Y
Enter student's name: Bob
Enter student's grade on midterm exam: 82
Enter student's grade on final exam: 85
Do you want to continue (Y/N)? N

NAME    GRADE
Alice   A
Bob     B
```

Inheritance

- Inheritance allows us to define a modified version of an existing class (superclass, parent class, or base class)
  - The new class is called the subclass, child class, or derived class
- Subclass inherits properties and methods of its superclass

- It can have its own properties and methods overriding some of the superclass' methods
- No initializer method is needed if the child class does not have its own properties (i.e., instance variables)

Example: A Semester Grade Class

```
class Student:  # Superclass
    def __init__(self, name="", midterm=0, final=0):
        self._name = name
        self._midterm = midterm
        self._final = final

    def setName(self, name):
        self._name = name

    def setMidterm(self, midterm):
        self._midterm = midterm

    def setFinal(self, final):
        self._final = final

    def getName(self):
        return self._name

    def __str__(self):
        return self._name + "\t" + self.calcSemGrade()

class LGstudent(Student): # Subclass of Student
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"
```

```
class PFstudent(Student): # Subclass of Student
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 60:
            return "Pass"
        else:
            return "Fail"
```

- The following function creates a list of both types of students and uses the list to display the names of the students and their semester grades

```
import student

def main():
    # students and grades
    listOfStudents = obtainListOfStudents()
    displayResults(listOfStudents)

def obtainListOfStudents():
    listOfStudents = []
    carryOn = 'Y'
    while carryOn == 'Y':
        name = input("Enter student's name: ")
        midterm = float(input("Enter grade on midterm: "))
        final = float(input("Enter grade on final: "))
        category = input("Enter category (LG or PF): ")
        if category.upper() == "LG":
            st = student.LGstudent(name, midterm, final)
        else:
            st = student.PFstudent(name, midterm, final)
        listOfStudents.append(st)
        carryOn = input("Do you want to continue (Y/N)? ")
        carryOn = carryOn.upper()
    return listOfStudents

def displayResults(listOfStudents):
    print("\nNAME\tGRADE")
    # Sort students by name.
    listOfStudents.sort(key = lambda x: x.getName())
    for pupil in listOfStudents:
        print(pupil)

main()
```

[Run]

```
Enter student's name: Bob
Enter grade on midterm: 79
Enter grade on final: 85
Enter category (LG or PF): LG
Do you want to continue (Y/N)? Y
Enter student's name: Alice
Enter grade on midterm: 92
Enter grade on final: 96
Enter category (LG or PF): PF
Do you want to continue (Y/N)? Y
Enter student's name: Carol
Enter grade on midterm: 75
Enter grade on final: 76
Enter category (LG or PF): LG
Do you want to continue (Y/N)? N

NAME    GRADE
Alice   Pass
Bob     B
Carol   C
```

"is-a" Relationship

- Child classes are specializations of their parent's class
  - Have all the characteristics of their parents
  - But, more functionality
  - Each child satisfies the "is-a" relationship with the parents
- E.g., each letter-grade student *is a* student, and each pass-fail student *is a* student

The *isinstance* Function

- A statement of the form

```
isinstance(object, className)
```

returns **True** if **object** is an instance of the named class or any of its subclasses, and otherwise returns **False**

- Some expressions involving the **isinstance** function

Expression	Value	Expression	Value
isinstance("Hello", str)	True	isinstance((), tuple)	True
isinstance(3.4, int)	False	isinstance({'b':"be"}, dict)	True
sinstance(3.4, float)	True	isinstance({}, dict)	True
isinstance([1, 2, 3], list)	True	isinstance({1, 2, 3}, set)	True
isinstance([], list)	True	isinstance({}, set)	False
isinstance((1, 2, 3), tuple)	True	isinstance(set(), set)	True

- The following function is an extension of the **displayResults** function on page 99
  - The **isinstance** function is used to count the number of letter-grade students

```
def displayResults(listOfStudents):
    print("\nNAME\tGRADE")
    numberOfLGstudents = 0
    listOfStudents.sort(key = lambda x: x.getName())
    for pupil in listOfStudents:
        print(pupil)
        # Keep track of number of letter-grade students.
        if isinstance(pupil, student.LGstudent):
            numberOfLGstudents += 1
    # Display number of students in each category.
    print("Number of letter-grade students:",
          numberOfLGstudents)
    print("Number of pass-fail students:",
          len(listOfStudents) - numberOfLGstudents)

main()
```

[Run]

```
NAME    GRADE
Alice   Pass
```

```
Bob     B
Carol   C
Number of letter-grade students: 2
Number of pass-fail students: 1
```

Adding New Instance Variables to a Subclass

- Child classes can also add properties (i.e., instance variables)
- Child class must contain an initializer method
  - Draws in the parent's properties
  - Then adds its own new properties
- The parameter list in the header of the child's initializer method should begin with **`self`**, list the parent's parameters, and add on its own new parameters
- The first line of the block should have the form

  `super().__init__(parentPar1, . . . , parentParN)`
- This line should be followed by standard declaration statements for the new parameters of the child

```
class PFstudent(Student):
    # A new Boolean parameter fullTime is added
    def __init__(self, name="", midterm=0, final=0,
                 fullTime=True):
        super().__init__(name, midterm, final)
        self._fullTime = fullTime

    def setFullTime(self, fullTime):
        self._fullTime = fullTime

    def getFullTime(self):
        return self._fullTime
```

```
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 60:
            return "Pass"
        else:
            return "Fail"

    def __str__(self):
        if self._fullTime:
            status = "Full-time student"
        else:
            status = "Part-time student"
        return (self._name + "\t" + self.calcSemGrade()
                + "\t" + status)
```

- The following program uses new definition of **`PFstudent`** on page 102

```
import studentWithStatus  # Contains new PFstudent definition

def main():
    ## Calculate and display a student's semester letter grade
    ## and status. Obtain student's name, grade on midterm
    ## exam, and grade on final.
    name = input("Enter student's name: ")
    midterm = float(input("Enter grade on midterm: "))
    final = float(input("Enter grade on final: "))
    category = input("Enter category (LG or PF): ")
    if category.upper() == "LG":
        st = studentWithStatus.LGstudent(name, midterm, final)
    else:
        question = input("Is " + name
                         + " a full time student (Y/N)? ")
        if question.upper() == 'Y':
            fullTime = True
        else:
            fullTime = False
        st = studentWithStatus.PFstudent(name, midterm,
                                         final, fullTime)
    # Display student's name, semester letter grade, and
    # status.
    print("\nNAME\tGRADE\tSTATUS")
    print(st)

main()
```

[Run]

```
Enter student's name: Alice
Enter grade on midterm: 92
Enter grade on final: 96
Enter category (LG or PF): PF
Is Alice a full time student (Y/N)? N

NAME    GRADE   STATUS
Alice   Pass    Part-time student
```

Overriding a Method

- If a method defined in the subclass has the same name as a method in its superclass, the child's method will override the parent's method
- Instead of the three classes **`student`**, **`LGstudent`**, and **`PFstudent`** as defined on p.98, the following program has only two classes, **`LGstudent`** and its subclass **`PFstudent`**
  - New definition is shorter and easier to read

```
def main():
    # Students and grades
    listOfStudents = obtainListOfStudents()
    displayResults(listOfStudents)

def obtainListOfStudents():
    listOfStudents = []
    carryOn = 'Y'
    while carryOn == 'Y':
        name = input("Enter student's name: ")
        midterm = float(input("Enter grade on midterm: "))
        final = float(input("Enter grade on final: "))
        category = input("Enter category (LG or PF): ")
        if category.upper() == "LG":
            st = LGstudent(name, midterm, final)
        else:
            st = PFstudent(name, midterm, final)
        listOfStudents.append(st)
        carryOn = input("Do you want to continue (Y/N)? ")
        carryOn = carryOn.upper()
    return listOfStudents
```

```
def displayResults(listOfStudents):
    print("\nNAME\tGRADE")
    listOfStudents.sort(key = lambda x: x.getName())
    for pupil in listOfStudents:
        print(pupil)

class LGstudent:
    def __init__(self, name="", midterm=0, final=0):
        self._name = name
        self._midterm = midterm
        self._final = final

    def setName(self, name):
        self._name = name

    def setMidterm(self, midterm):
        self._midterm = midterm

    def setFinal(self, final):
        self._final = final

    def getName(self):
        return self._name

    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"

    def __str__(self):
        return self._name + "\t" + self.calcSemGrade()

class PFstudent(LGstudent):
    def calcSemGrade(self):
        average = round((self._midterm + self._final) / 2)
        if average >= 60:
            return "Pass"
        else:
            return "Fail"

main()
```

Polymorphism

- A feature of all object-oriented programming languages
- Allows two classes to use the same method name but with different implementations
  - **`calcSemGrade`** on pages 98 and 105

Multiple Inheritance

- A class can be derived from more than one base class
  - The features of all the base classes are inherited into the derived class

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

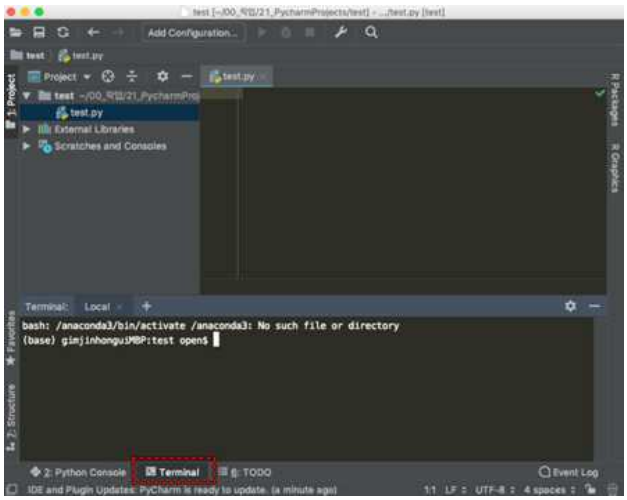
- Method resolution order (MRO):
  - Any specified attribute is searched first in the current class
  - If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice

08 NumPy and Matplotlib Packages

Installation

- Open the Command Prompt from PyCharm and use pip (Package Installer for Python):

```
> pip install numpy

> pip install matpolotlib
```



NumPy

- NumPy stands for 'Numerical Python'
  - A library consisting of multidimensional array objects and a collection of routines for processing of array
- Using NumPy, a developer can perform the following:
  - Mathematical and logical operations on arrays
  - Fourier transforms and routines for shape manipulation

- Operations related to linear algebra
- Random number generation

Arrays

- The most important object defined in NumPy is the n-dimensional array type called ndarray
  - The elements of the array all have the same type
  - Items in the collection can be accessed using a zero-based index

```
import numpy as np

# One-dimensional array
a = np.array([1, 2, 3])
# Two-dimensional array
b = np.array([[1, 2], [3, 4]])
# To specify minimum number of dimensions
c = np.array([1, 2, 3, 4, 5], ndmin = 2)
# To specify desired data type for the array
d = np.array([1, 2, 3], dtype = float)
print(a)
print(b)
print(c)
print(d)
print(a[2])
a
b
```

[Run]

```
[1 2 3]
[[1 2]
 [3 4]]
[[1 2 3 4 5]]
[1. 2. 3.]
3
array([1, 2, 3])
array([[1, 2],
       [3, 4]])
```

Array Creation

- A new ndarray object can be constructed by any of the following array creation routines

```
import numpy as np

a = np.arange(5)        # Array of range(5)
b = np.arange(3, 9, 2) # From 3 to 9 in increments of 2
c = np.zeros(5) # Array of five zeros (default type is float)
d = np.ones(5)  # Array of five ones (default type is float)
print(a)
print(b)
print(c)
print(d)
```

[Run]

```
[0 1 2 3 4]
[3 5 7]
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
```

```
import numpy as np

a = np.eye(3)      # Produces square identity matrix
b = np.eye(3, 4)  # Identity matrix of size 3 × 4
# To produce an array of 5 elements equally spaced from
# 10 to 20
c = np.linspace(10, 20, 5)

print(a)
print(b)
print(c)
```

[Run]

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]]
[10.  12.5 15.  17.5 20. ]
```

Getting Information and Changing Shapes

- We can get various information about an array and change its shape in a variety of ways

```
import numpy as np

a = np.arange(6)
print(a)
b = a.reshape(3, 2) # Reshaped to 3 × 2
print(b)
print(b[2, 1])
print(b[:, 1])
print(np.shape(b)) # Array dimension as a tuple
print(np.size(b))  # Total number of elements
print(np.ndim(b))  # Number of dimensions
print(np.transpose(b))  # Computes matrix transpose

c = np.copy(b)      # Makes a deep copy
print(c)
print(np.ravel(c)) # Makes the array one-dimensional
```

[Run]

```
[0 1 2 3 4 5]
[[0 1]
 [2 3]
 [4 5]]
5
[1 3 5]
(3, 2)
6
2
[[0 2 4]
 [1 3 5]]
[[0 1]
 [2 3]
 [4 5]]
[0 1 2 3 4 5]
```

Mathematical Functions

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[11, 12], [13, 14]])
```

```
print(a + b)           # matrix addition
print(a * b)           # element-wise multiplication
print(np.dot(a, b))    # matrix multiplication
print(np.inner(a, b)) # inner product of row vectors
```

[Run]

```
[[12 14]
 [16 18]]
[[11 24]
 [39 56]]
[[37 40]
 [85 92]]
[[35 41]               # [[(1, 2)·(11, 12) (1, 2)·(13, 14)]
 [81 95]]              #  [(3, 4)·(11, 12) (3, 4)·(13, 14)]]
```

```
import numpy as np

a = np.array ([-1.7, 1.5, -0.2, 0.6, 10])
b = np.around(a)
c = np.floor(a)
d = np.ceil(a)

print(b)
print(c)
print(d)
```

[Run]

```
[-2.  2. -0.  1. 10.]
[-2.  1. -1.  0. 10.]
[-1.  2. -0.  1. 10.]
```

```
import numpy as np

a = np.array([1.0, 5.55, 123, 0.567, 25.532])
b = np.around(a)
c = np.around(a, decimals = 1)
d = np.around(a, decimals = -1)  # default is 0

print(b)
print(c)
print(d)
```

[Run]

```
[  1.   6. 123.   1.  26.]
[  1.    5.6 123.    0.6  25.5]
[  0.  10. 120.   0.  30.]
```

Statistical Functions

```
import numpy as np

a = np.array([[3,7,5],[8,4,3],[2,4,9]])
b = np.amin(a, 1) # Minimum element along the rows (axis=1)
c = np.amin(a, 0) # Minimum element along the cols (axis=0)
d = np.amin(a)    # Minimum element in the given array
e = np.amax(a, 1)
f = np.amax(a, 0)
print(b)
print(c)
print(d)
print(e)
print(f)
```

[Run]

```
[3 3 2]
[2 4 3]
2
[7 8 9]
[8 7 9]
```

```
import numpy as np

a = np.array([1,2,3,4])
b = np.average(a)
c = np.std(a)    # Standard deviation
d = np.var(a)    # Variance

print(b)
print(c)
print(d)
```

[Run]

```
2.5
1.118033988749895
1.25
```

Random Numbers

```
import numpy as np

# To produce a 3 × 2 matrix of random numbers uniformly
# distributed between 0 and 1
b = np.random.rand(3,2)
```

```
# To produce a 2 × 4 matrix of random integers uniformly
# distributed in range(3, 9)
c = np.random.randint(3, 9, size=(2,4))

print(b)
print(c)
```

[Run]

```
[[0.95248403 0.96327307]
 [0.21416881 0.83092589]
 [0.59329024 0.25912892]]
[[5 3 3 5]
 [3 7 8 8]]
```

Linear Algebra

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -4 \\ 1 \end{bmatrix}$$

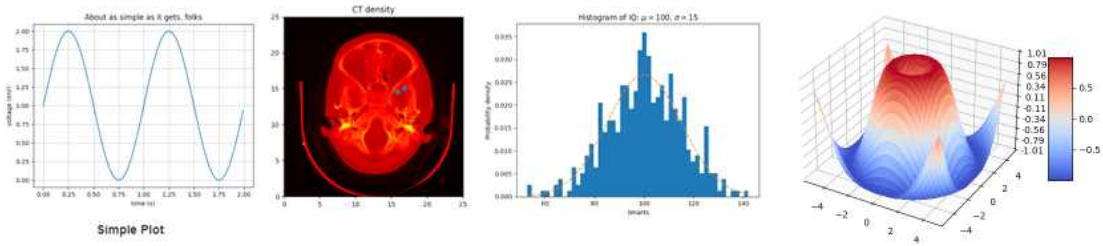
```
import numpy as np

a = np.array([[1,2],[3,4]])
b = np.array([[-4],[1]])
inv_a = np.linalg.inv(a)   # inverse of a
c = np.linalg.solve(a, b)  # compute x such that ax=b
d = np.linalg.det(a)  # determinant of a
e = np.linalg.eig(a)  # eigenvalues and eigenvectors of a

print(inv_a)
print(c)
print(d)
print(e)
```

[Run]

```
[[-2.   1. ]
 [ 1.5 -0.5]]
[[ 9. ]
 [-6.5]]
-2.0000000000000004
(array([-0.37228132, 5.37228132]),
 array([[-0.82456484, -0.41597356],
        [0.56576746, -0.90937671]]))
```

Matplotlib

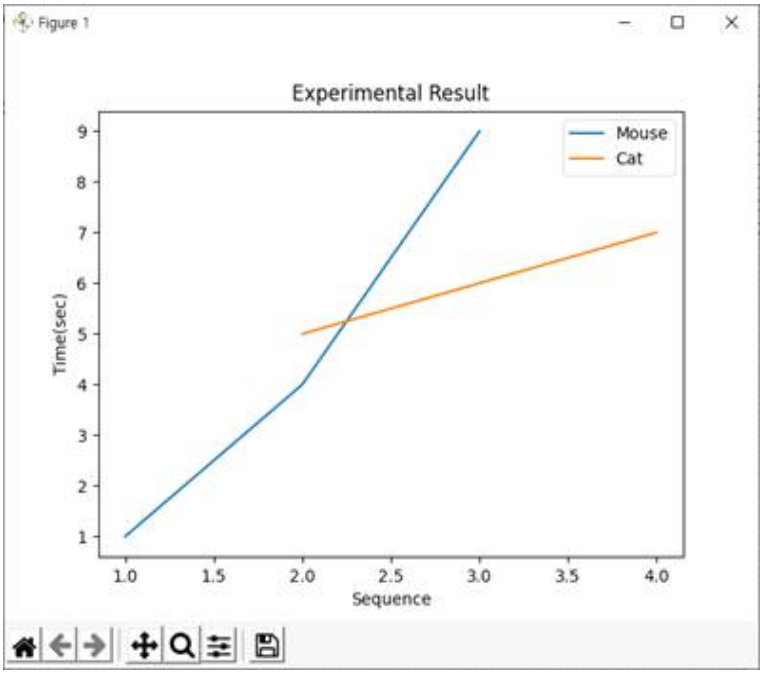
- Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy
  - Provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits
  - Can handle line plot, images, histograms, bar/pie charts, tables, scatter plots, etc.



Pyplot Module

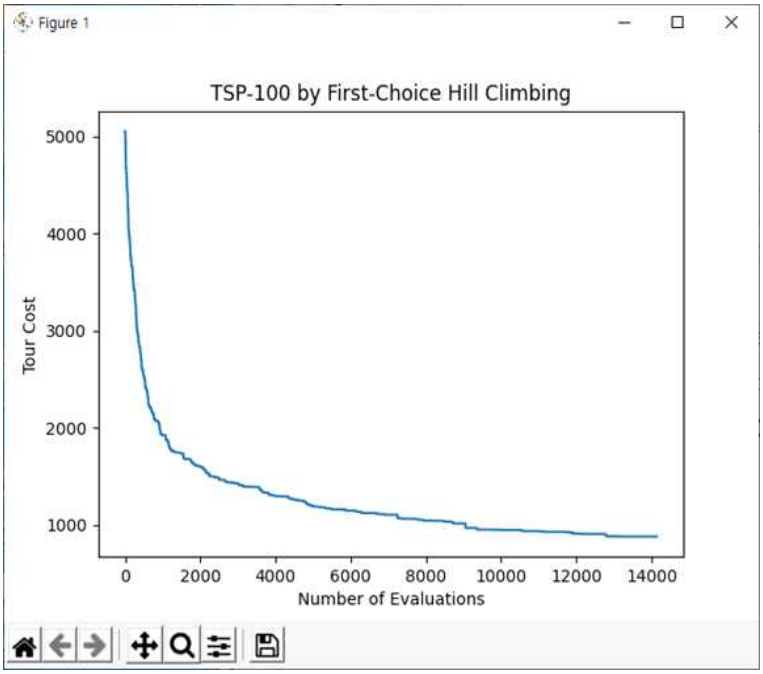
```
from matplotlib import pyplot as plt

plt.plot([1, 2, 3], [1, 4, 9]) # Values for x and y axes
plt.plot([2, 3, 4], [5, 6, 7]) # Should be of the same lengths
plt.xlabel('Sequence')
plt.ylabel('Time(sec)')
plt.title('Experimental Result')
plt.legend(['Mouse', 'Cat'])
plt.show()
```

[Run]



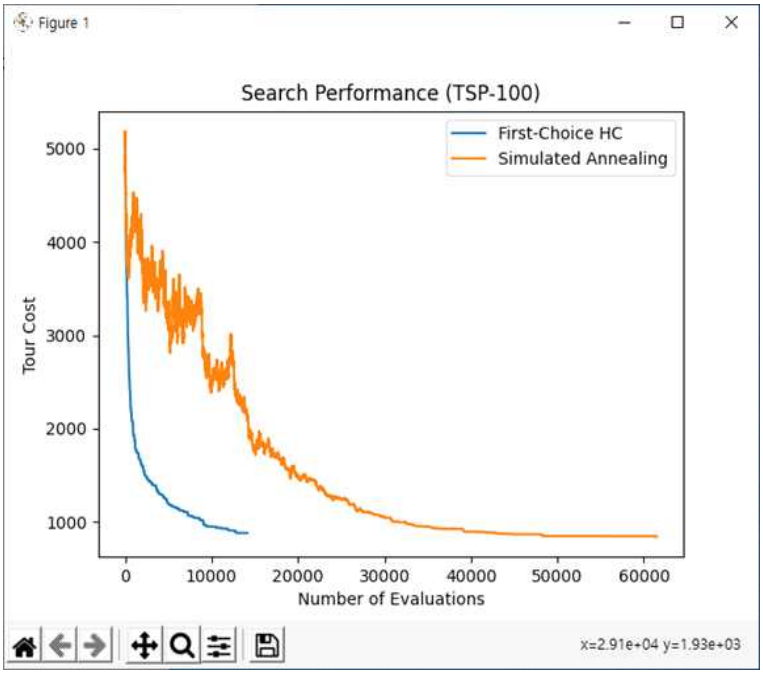

```
import numpy as np
import matplotlib.pyplot as plt

# Search performance by First-choice hill climbing for TSP

x = np.arange(14140) # There are 14140 values to be plotted

y = []  # Y values will be read in from a file
infile = open("firstChoice.txt", 'r')  # Result file
for line in infile:
    y.append(float(line))
infile.close()

plt.plot(x, y)
plt.xlabel('Number of Evaluations')
plt.ylabel('Tour Cost')
plt.title('TSP-100 by First-Choice Hill Climbing')
plt.show()
```

[Run]



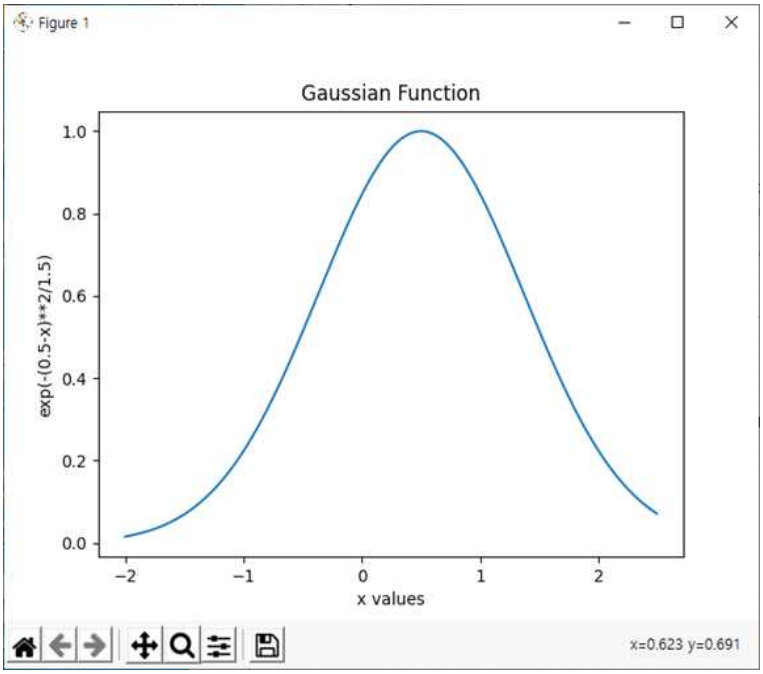

```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.arange(14140)  # First-choice hill climbing
y1 = []
infile = open("firstChoice.txt", 'r')
for line in infile:
    y1.append(float(line))
infile.close()

x2 = np.arange(61500)  # Simulated annealing
y2 = []
infile = open("SA.txt", 'r')
for line in infile:
    y2.append(float(line))
infile.close()

plt.plot(x1, y1)
plt.plot(x2, y2)
plt.xlabel('Number of Evaluations')
plt.ylabel('Tour Cost')
plt.title('Search Performance (TSP-100)')

# Legend elements are in the order of the above plots
plt.legend(['First-Choice HC', 'Simulated Annealing'])

plt.show()
```

[Run]




```
import numpy as np
import matplotlib.pyplot as plt

# Plot for a Gaussian function

def gaussian(x):
    return np.exp(-(0.5 - x)**2/1.5)

x = np.arange(-2, 2.5, 0.01)
y = gaussian(x)  # Array argument produces array output

plt.plot(x, y)
plt.xlabel('x values')
plt.ylabel('exp(-(0.5-x)**2/1.5)')
plt.title('Gaussian Function')
plt.show()
```

[Run]



- The previous program can be rewritten by using **`lamda`** function instead of the **`gaussian`** function

```
import numpy as np
import matplotlib.pyplot as plt

# Plot for a Gaussian function

gaussian = lambda x: np.exp(-(0.5 - x)**2/1.5)

x = np.arange(-2, 2.5, 0.01)
y = gaussian(x)  # Array argument produces array output

plt.plot(x, y)
plt.xlabel('x values')
plt.ylabel('exp(-(0.5-x)**2/1.5)')
plt.title('Gaussian Function')
plt.show()
```

Homework #1

Note: For the following exercises, write a program to solve the problem and display the answer. A possible output is shown in a shaded box and responses to input statements appear underlined.

Exercises 2.2

110. Word Replacement Write a program that requests a sentence, a word in the sentence, and another word and then displays the sentence with the first word replaced by the second. Do not use the format method.

```
Enter a sentence: What you don't know won't hurt you.
Enter word to replace: know
Enter replacement word: owe
What you don't owe won't hurt you.
```

Exercises 2.3

58. Present Value The present value of f dollars at interest rate r% compounded annually for n years is the amount of money that must be invested now in order to grow to f dollars (called the future value) in n years where the interest rate is r% per year. The formula for present value is

$$present\ value = \frac{f}{\left(1 + \frac{r}{100}\right)^n}$$

Calculate the present value of an investment after the user enters the future value, interest rate, and number of years. The following figure shows that at 4% interest per year, $7,903.15 must be invested now in order to have

$10,000 after 6 years. The program should use the format method to display the outputs.

```
Enter future value: 10000
Enter interest rate (as %): 4
Enter number of years: 6
Present value: $7,903.15
```

Exercises 3.2

32. Pig Latin Write a program that requests a word (in lowercase letters) as input and translates the word into Pig Latin. The rules for translating a word into Pig Latin are as follows:

(a) If the word begins with a group of consonants, move them to the end of the word and add *ay*. For instance, *chip* becomes *ipchay*.

(b) If the word begins with a vowel, add *way* to the end of the word. For instance, *else* becomes *elseway*.

```
Enter word to translate: chip
The word in Pig Latin is ipchay.
```

Exercises 3.3

16. Bouncing Ball The coefficient of restitution of a ball, a number between 0 and 1, specifies how much energy is conserved when the ball hits a rigid surface. A coefficient of .9, for instance, means a bouncing ball will rise to 90% of its previous height after each bounce. Write a program to input a coefficient of restitution and an initial height in meters, and report how many times a ball bounces when dropped from its initial height before it rises to a

height of less than 10 centimeters. Also report the total distance traveled by the ball before this point. The coefficients of restitution of a tennis ball, basketball, super ball, and softball are .7, .75, .9, and .3, respectively.

```
Enter coefficient of restitution: .7
Enter initial height in meters: 8
Number of bounces: 13
Meters traveled: 44.82
```

Exercises 3.4

82. Digit Sum Write a program to calculate the total sum of the digits in the integers from 1 to a million.

```
The sum of the digits in the numbers
from 1 to one million is 27,000,001.
```

Homework #2

Note: For each of the following exercises, write a program to carry out the stated task. A possible output is shown in a shaded box and responses to input statements appear underlined.

Exercises 2.4

102. Analyze a Sentence Write a program that displays the first and last words of a sentence input by the user. Assume that the only punctuation is a period at the end of the sentence.

```
Enter a sentence: Reach for the stars.
First word: Reach
Last word: stars
```

104. Name Write a program that requests a three-part name and then displays the middle name.

```
Enter a 3-part name: Michael Andrew Fox
Middle name: Andrew
```

Exercises 3.4

66. Median The median of an ordered set of measurements is a number separating the lower half from the upper half. If the number of measurements is odd, the median is the middle measurement. If the number of measurements is even, the median is the average of the two middle measurements. Write a program that requests a number n and a set of n measurements (not necessarily

ordered) as input and then displays the median of the measurements.

```
How many numbers do you want to enter? 4
Enter a number: 9
Enter a number: 3
Enter a number: 6
Enter a number: 5
Median: 5.5
```

74. A Puzzle The following puzzle is known as *The Big Cross-Out Swindle*. "Beginning with the word 'NAISNIENLGELTETWEORRSD,' cross out nine letters in such a way that the remaining letters spell a single word". Write a program that creates variables named *startingWord*, *crossedOutLetters*, and *remainingLetters*. The program should assign to *startingWord* the string given in the puzzle, assign to *crossedOutLetters* a list containing every other letter of *startingWord* beginning with the initial letter *N*, and assign to *remainingLetters* a list containing every other letter of *startingWord* beginning with the second letter, *A*. The program should then display the values of the three variables.

```
Starting word: NAISNIENLGELTETWEORRSD
Crossed out letters: N I N E L E T T E R S
Remaining letters: A S I N G L E W O R D
```

78. Special Number Write a program to find the four-digit number, call it *abcd*, whose digits are reversed when the number is multiplied by 4. That is, $4 \times abcd = dcba$.

```
Since 4 times 2178 is 8712,
the special number is 2178.
```

Exercises 4.1

26. Count Function Suppose the count function for a string didn't exist. Define a function that returns the number of non-overlapping occurrences of a substring in a string.

30. Pay Raise Write a pay-raise program that requests a person's first name, last name, and current annual salary, and then displays the person's salary for next year. People earning less than $40,000 will receive a 5% raise, and those earning $40,000 or more will receive a raise of $2,000 plus 2% of the amount over $40,000. Use functions for input and output, and a function to calculate the new salary.

```
Enter first name: John
Enter last name: Doe
Enter current salary: 48000
New salary for John Doe: $50,160.00
```

Homework #3

Note: For each of the following exercises, write a program to carry out the stated task. A possible output is shown in a shaded box and responses to input statements appear underlined.

Exercises 4.2

68. Mortgage Calculations Write a program to calculate three monthly values associated with a mortgage. The interest paid each month is the monthly rate of interest (annual rate of interest / 12) applied to the balance at the beginning of the month. Each month the reduction of principal equals the monthly payment minus the interest paid. At any time, the balance of the mortgage is the amount still owed—that is, the amount required to pay off the mortgage. The end of month balance is calculated as [beginning of month balance] - [reduction of principal]. The main function should call three functions—one (multi-valued) for input, one (multi-valued) to calculate the values, and one for output.

```
Enter annual rate of interest: 5
Enter monthly payment: 1932.56
Enter beg. of month balance: 357819.11
Interest paid for the month: $1,490.91
Reduction of principal: $441.65
End of month balance: $357,377.46
```

70. Wilson's Theorem A number is prime if its only factors are 1 and itself. Write a program that determines whether a number is prime by using the theorem "The number n is a prime number if and only if n divides

(n – 1)! + 1." The program should define a Boolean-valued function named *isPrime* that calls a function named *factorial*.

```
Enter an integer greater than 1: 37
37 is a prime number.
```

CHAPTER 4 PROGRAMMING PROJECTS

5. Alphabetical Order The following words have three consecutive letters that are also consecutive letters in the alphabet: THI**RST**Y, **NOP**E, A**FGH**ANISTAN, **STU**DENT. Write a program that accepts a word as input and determines whether or not it has three consecutive letters that are consecutive letters in the alphabet. The program should use a Boolean-valued function named *isTripleConsecutive* that accepts an entire word as input. ***Hint:*** Use the **`ord`** function.

```
Enter a word: HIJACK
HIJACK contains three successive letters
in consecutive alphabetical order.
```

Exercises 5.1

40. Months The file **`SomeMonths.txt`** initially contains the names of the 12 months, one per line. Write a program that deletes all months from the file that do not contain the letter *r*.

46. File of Names The file **`Names.txt`** contains a list of first names in alphabetical order. Write a program that requests a name from the user and

inserts the name into the file in its proper location. If the name is already in the file, it should not be inserted. You should use set operations in your program.

CHAPTER 5 PROGRAMMING PROJECTS

1. Unit Conversions The following table contains some lengths in terms of feet. Write a program that displays the nine different units of measure; requests the unit to convert from, the unit to convert to, and the quantity to be converted; and then displays the converted quantity. A typical outcome is shown in the shaded box below. Use the file **`Units.txt`** to create a dictionary that provides the number of feet for a given unit of length. The first two lines of the file are `inches,.083333; furlongs,660.`

Equivalent lengths.

1 inch = .083333 foot	1 rod = 16.5 feet
1 yard = 3 feet	1 furlong = 660 feet
1 meter = 3.28155 feet	1 kilometer = 3281.5 feet
1 fathom = 6 feet	1 mile = 5280 feet

```
UNITS OF LENGTH
inches      feet        miles
meters      fathoms     yards
kilometers  furlongs    rods

Unit to convert from: yards
Unit to convert to: miles
Enter length in yards: 555
Length in miles: 0.3153
```

Homework #4

Note: For each of the following exercises, write a program to carry out the stated task. A possible output is shown in a shaded box and responses to input statements appear underlined.

Exercises from David I. Schneider

Exercises 6.1

29. The following program will perform properly if the user enters 0 in response to the request for input. However, the program will crash if the user responds with "eight". Rewrite the program using a **try/ except** statement so that it will handle both types for responses.

```
while True:
    n = int(input("Enter a nonzero integer: "))
    if n != 0:
        reciprocal = 1 / n
        print("The reciprocal of {0} is {1:,.3f}"\
        .format(n, reciprocal))
        break
    else:
        print("You entered zero. Try again.")
```

```
Enter a nonzero integer: 0
You entered zero. Try again.
Enter a nonzero integer: eight
You did not enter a nonzero integer. Try again.
Enter a nonzero integer: 8
The reciprocal of 8 is 0.125
```

31. Enter a Number Write a robust program that requests an integer from 1 through 100.

```
Enter an integer from 1 to 100: 5.5
You did not enter an integer.
Enter an integer from 1 to 100: five
You did not enter an integer.
Enter an integer from 1 to 100: 555
Your number was not between 1 and 100.
Enter an integer from 1 to 100: 5
Your number is 5.
```

Exercises 6.2

Write lines of code to carry out the stated task. Note that the *random* module has to be imported.

9. Alphabet Display three letters selected at random from the alphabet.

13. Coin Toss a coin 100 times and display the number of times that a "Head" occurs.

17. Matching Cards Suppose two shuffled decks of cards are placed on a table, and then cards are drawn from the tops of the decks one at a time and compared. On average, how many matches do you think will occur? Write a program to carry out this process 10,000 times and calculate the average number of matches that occur. A possible output is shown in the shaded box below.

```
The average number of cards
that matched was 1.013.
```

Exercises 6.4

7. Alphabetical Order The following iterative function determines whether a list of lowercase words is in alphabetical order. Write the equivalent recursive function.

```
def isAlpha(L):
    ## Determine whether list of lowercase words is in
    ## alphabetical order.
    for i in range(len(L) - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

8. Sequence of Numbers The following iterative function displays a sequence of numbers. Write the equivalent recursive function.

```
def displaySequenceOfNumbers(m, n):
    ## Display the numbers from m to n, where m <= n.
    while m <= n:
        print(m)
        m = m + 1
```

13. Reverse Order Write a program that asks the user to input an arbitrary number of names of states, and then displays the names in the reverse order they were entered. Do not use lists or files to store the names. A possible

output is shown in the shaded box below and responses to input statements appear underlined.

```
Enter a state: Ohio
Enter a state: Texas
Enter a state: Oregon
Enter a state: End
Oregon
Texas
Ohio
```

CHAPTER 6 PROGRAMMING PROJECTS

3. Analyze a Bridge Hand Write a program that randomly selects and displays 13 cards from the deck of cards and gives the suit distribution. The binary file **`DeckOfCardsList.dat`** contains the deck of cards in the form of a list of strings, where each string represents a card as e.g., '7♣'. A possible output is shown in the shaded box below.

```
5♠, K♠, 5♥, J♥, A♣, 4♣, 5♣, J♣, 6♥, 10♣, 2♣, 6♠, 7♠
Number of ♠ is 4
Number of ♥ is 3
Number of ♣ is 6
```

Homework #5

Exercises fromDavid I. Schneider

Exercises 7.1

27. Fraction Create a class named *Fraction*having instance variables for numerator and denominator, and a method that reduces a faction to lowest terms by dividing the numerator and denominator by their greatest common divisor. The code for obtaining the greatest common divisor of two nonzero integers *m*and *n*is given below. Save the class in the file **`fraction.py`**.

Note:This class will be used in Exercises 28 and 29.

```
def GCD(m, n):  #Greatest Common Divisor
    while n != 0:
        t = n
        n = m % n
        m = t
    return m
```

28. Reduce a Fraction Write a program that requests a fraction as input and reduces the fraction to lowest terms. A possible output is shown in the shaded box below. Use the class *Fraction*created in Exercise 27.

```
Enter numerator of fraction: 12
Enter denominator of fraction: 30
Reduction to lowest terms: 2/5
```

29. Convert a Decimal to a Fraction Write a program that converts a decimal number to an equivalent fraction. A possible output is shown in the shaded box below. Use the class *Fraction*created in Exercise 27.

```
Enter a positive decimal number less than 1: .375
Converted to fraction: 3/8
```

32. Quiz Grades An instructor gives six quizzes during a semester with quiz grades 0 through 10, and drops the lowest grade. Write a program to find the average of the remaining five grades. A possible output is shown in the shaded box below. The program should use a class named *Quizzes*that has an instance variable to hold a list of the six grades, a method named *average*, and a *__str__*method.

```
Enter grade on quiz 1: 9
Enter grade on quiz 2: 10
Enter grade on quiz 3: 5
Enter grade on quiz 4: 8
Enter grade on quiz 5: 10
Enter grade on quiz 6: 10
Quiz average: 9.4
```

Exercises 7.2

11. Rock, Paper, Scissors Write a program to play a three-game match of "rock, paper, scissors" between a person and a computer. A possible output is shown in the shaded box below, where the last line should be changed to "TIE" in case of a tie. The program should use a class named *Contestant*having two subclasses named *Human*and *Computer*. After the person makes his or her choice, the computer should make its choice at random. The

*Contestant*class should have instance variables for *name*and *score*.

```
Enter name of human: Garry
Enter name of computer: Big Blue

Garry, enter your choice: rock
Big Blue chooses paper
Garry: 0 Big Blue: 1

Garry, enter your choice: scissors
Big Blue chooses rock
Garry: 0 Big Blue: 2

Garry, enter your choice: paper
Big Blue chooses scissors
Garry: 0 Big Blue: 3

BIG BLUE WINS
```

CHAPTER 7 PROGRAMMING PROJECTS

1. United Nations The file **`UN.txt`** gives data about the 193 members of the United Nations. Each line of the file contains four pieces of data about a country—*name*, *continent*, *population* (in millions), and *land area* (in square miles). Some lines of the file are

`Canada,North America,34.8,3855000`

`France,Europe,66.3,211209`

`New Zealand,Australia/Oceania,4.4,103738`

`Nigeria,Africa,177.2,356669`

`Pakistan,Asia,196.2,310403`

`Peru,South America,30.1,496226`

(a) Create a class named *Nation* with four instance variables to hold the data for a country and a method named *popDensity* that calculates the population density of the country. Write a program that uses the class to create a

dictionary of 193 items, where each item of the dictionary has the form

name of a country: Nation object for that country

Use the file **UN.txt** to create the dictionary, and save the dictionary in a pickled binary file named **nationsDict.dat**. Also save the class *Nation* in a file named **nation.py**.

(c) Write a program that requests the name of a continent as input, and then displays the names (in descending order) of the five most densely populated U.N. member countries in that continent. A possible output is shown in the shaded box below. Use the pickled binary file **nationsDict.dat** and the file **nation.py** created in part (a). To use the definition of the class *Nation*, you need to include the following statement at the beginning of your program

```
from nation import Nation
```

```
Enter a continent: South America
  Ecuador
  Colombia
  Venezuela
  Brazil
  Peru
```

Homework #6

Note: For each of the following exercises, write a program to solve the problem and check before the answer. The program should use variables for each of the quantities.

Exercises 1

a) Make an array `a` of size 6×4 where every element is a 2.

b) Make an array `b` of size 6×4 that has 3 on the leading diagonal and 1 everywhere else. (Do not use loops.)

c) Can you multiply these two arrays together? Then, explain why `a * b` works, but `dot(a, b)` does not.

d) Compute `dot(a.transpose(), b)` and `dot(a, b.transpose())`. Explain why the results are in different shapes?

Exercises 2

We have some data about House prices and parameters that affect the price.

Bias	Size	Number of bedrooms	Number of floors	Age of house	Price(1000000won)
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1544	3	2	30	315
1	852	2	1	36	178

a) Make an array X that has the same data as the above table (shape of X: (4, 6)). Check whether the shape of X is (4, 6) by using assert() function.

(Note: assert(cond)gives an error if cond is not True.)

b) Then, extract 4 parameters (size, # of bedrooms, floors, age of house) from X and put them into a new matrix P. Check if the shape of P is (4, 4) by using assert() function.

c) Find the mean value of each parameter using the numpy mean function.

Exercises 3

a) Create a 4′4 array a using np.ones, another 4′4 array b using np.eye, and a list c that have 4 elements each of which is a list of 4 numbers. Now, execute sum_ = a + b and sub = a − c and then calculate the mean of sum_. (Note: We use the variable name sum_ because sum is the name of a python library function.)

b) Define a function findOne that receives an array and counts the number of ones in it. Make a 5′5 array of random integers and count the number of ones in it using findOne. Do the same thing using the 'where()' function of NumPy. (Note: Use help(np.where) to find out how to use it.)

Exercises 4

a) Plot the sigmoid function $\sigma(x) = 1 / (1 + e^{-x})$ using `np.arange(-2, 2, 0.01)`.

b) You are given some experimental result data recorded in two text files "`tError.txt`" and "`vError.txt`", where there is one float number in each line. Plot both data on a single graph. The horizontal and vertical axes represent 'model complexity' and 'error rate', respectively.

09 Search Algorithms:

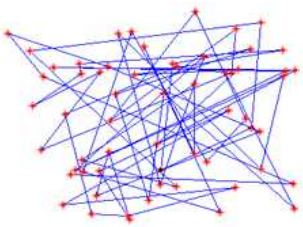
Object-Oriented Implementation (Part A)

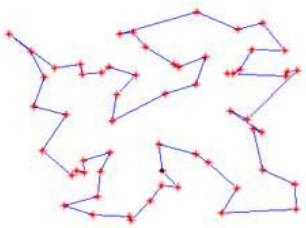
Conventional vs. AI Algorithms

- Intractable problems:
  - There are many optimization problems that require a lot of time to solve but no efficient algorithms have been identified
  - Exponential algorithms are useless except for very small problems

Example: Traveling Salesperson Problem:

The salesperson wants to minimize the total traveling cost required to visit all the cities in a territory, and return to the starting point





- Combinatorial Explosion: there are $n!$ routes to investigate

  → Efficiency is the issue!

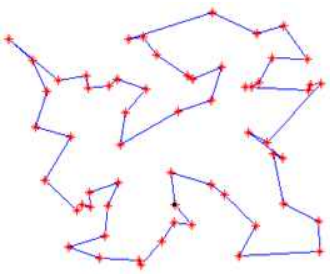
- Nearest neighbor heuristic: always goes to the nearest city
  - Given a starting city, there are $(n-1)+(n-2)+\cdots+1 = n(n-1)/2$ cases to consider
  - Since there are n different ways to start, the total number of cases is $n^2(n-1)/2$

- Can find a near optimal solution in a much shorter time

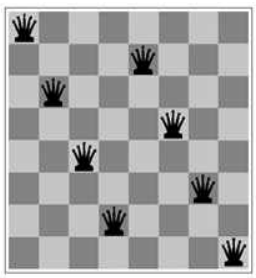
- Conventional algorithms (e.g., sorting algorithms) are often called exact algorithms because they always find a correct or an optimal solution
- AI algorithms use heuristics or randomized strategies to find a near optimal solution quickly

Local Search Algorithms

- Iterative improvement algorithms:
  - State space = set of "complete" configurations (e.g., complete tours in TSP)
  - Find optimal configuration according to the objective function
  - Find configuration satisfying constraints (e.g., n-queens Problem)
- Start with a complete configuration and make modifications to improve its quality



TSP



8-queens

Example: TSP

- Current configuration:

B	A	C	E	D

- Candidate neighborhood configurations:

B	E	C	A	D
B	A	E	C	D
E	C	A	B	D
B	A	C	D	E
C	A	B	E	D

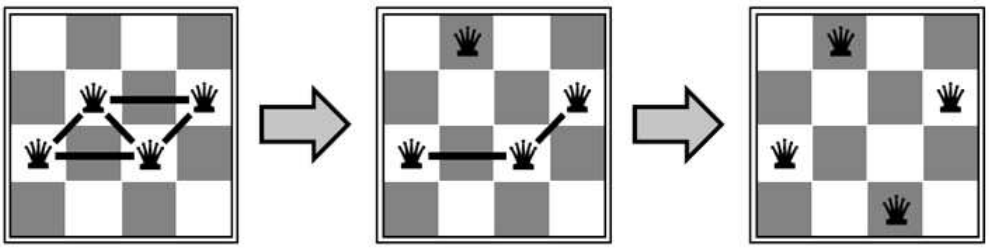
Mutation by inversion:

A random subsequence is inverted

Variants of this approach get within 1% of optimal very quickly with thousands of cities

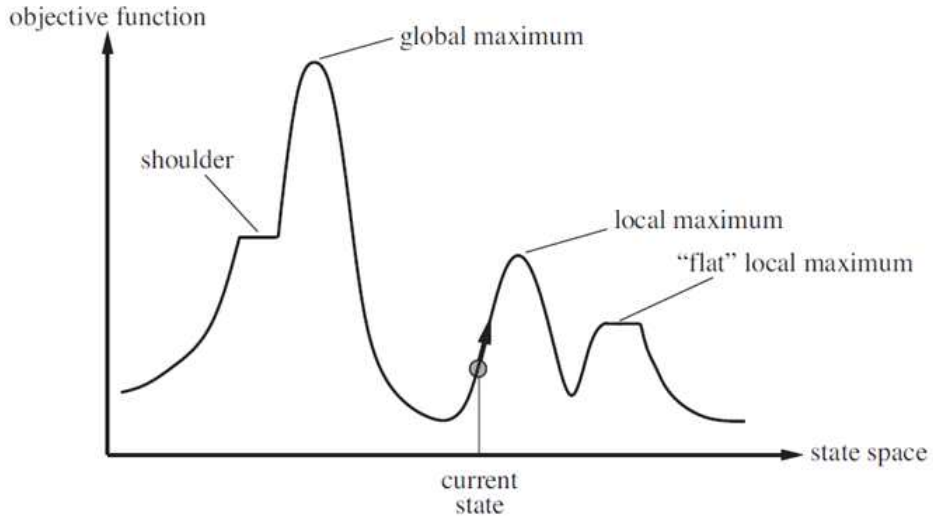
Example: n-queens

- Move a queen to reduce number of conflicts



- Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1$ million

- State space landscape
  - Location: state
  - Elevation: heuristic cost function or objective function



Hill-Climbing Search

- "Like climbing Everest in thick fog with amnesia"
  - Continually moves in the direction of increasing value
  - Also called gradient ascent/descent search

[Steepest ascent version]

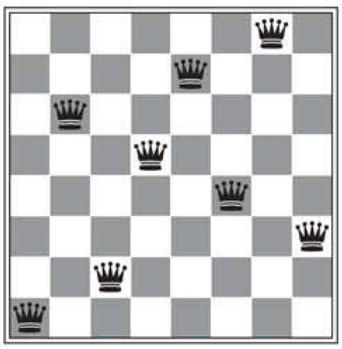
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Example: 8-queens problem

- Each state has 8 queens on the board, one per column

- Successor function generates 56 states by moving a single queen to another square in the same column
- h is the # of pairs that are attacking each other

A state with $h = 17$



A local minimum

Example: Suppose we want to site three airports in Romania:

- 6-D state space defined by (x_1, y_1), (x_2, y_2), (x_3, y_3)
- Objective function:

  $f(x_1, y_1, x_2, y_2, x_3, y_3)$ = sum of squared distances from each city to nearest airport

- We can discretize the neighborhood of each state and apply any local search algorithm
  - ○ Move in each direction by a fixed amount $\pm\delta$ (12 successors)
- We can directly (without discretization) search in continuous spaces
  - ○ Successors are chosen randomly by generating 6-dimensional random vectors of length δ

- Drawbacks: often gets stuck to local maxima due to greediness
- Possible solutions:

- Stochastic hill climbing:
  - ◯ Chooses at random from among the uphill moves with probability proportional to steepness
- First-choice (simple) hill climbing:
  - ◯ Generates successors randomly until one is found that is better than the current state
- Random-restart hill climbing:
  - ◯ Conducts a series of hill-climbing searches from randomly generated initial states
  - ◯ Very effective for 8-queens

    (Can find solutions for 3 million queens in under a minute)

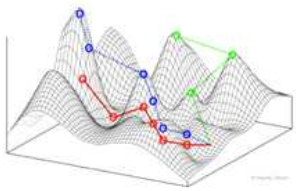
- Complexity:
  - The success of hill climbing depends on the shape of the state-space landscape
  - NP-hard problems typically have an exponential number of local maxima to get stuck on
  - A reasonably good local maximum can often be found after a small number of restarts

Continuous State Spaces

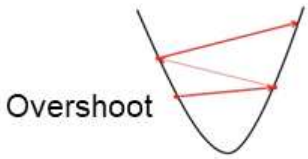
- Gradient methods attempt to use the gradient of the landscape to maximize/minimize f by

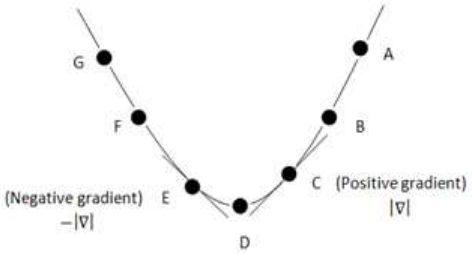
  $\boldsymbol{x} \leftarrow \boldsymbol{x}$ +/- $\alpha \nabla f(\boldsymbol{x})$ (α : update rate)

where $\nabla f(\boldsymbol{x})$ is the gradient vector (containing all of the partial derivatives) of f that gives the magnitude and direction of the steepest slope



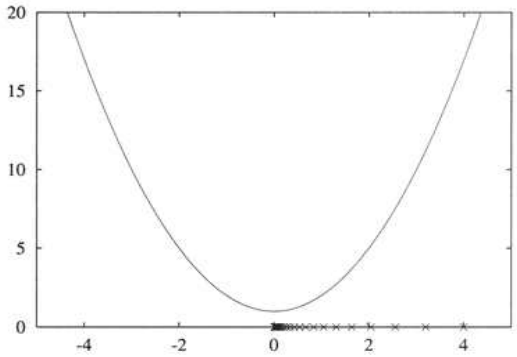
- Too small α : too many steps are needed
- Too large α : the search could overshoot the target
- Points where $\nabla f(\boldsymbol{x}) = 0$ are known as critical points





Example: Gradient descent

- If $f(w) = w^2 + 1$, then $f'(w) = 2w$ $\quad w \leftarrow w - \alpha f'(w)$
- Starting from an initial value $w = 4$, with the step size of 0.1:
  - ◯ $4 - (0.1 \times 2 \times 4) = 3.2$
  - ◯ $3.2 - (0.1 \times 2 \times 3.2) = 2.56$
  - ◯ $2.56 - (0.1 \times 2 \times 2.56) = 2.048$
  - . . . . . .



- Stops when the change in parameter value becomes too small

Simulated Annealing Search

- Idea:
  - Efficiency of valley-descending + completeness of random walk
  - Escape local minima by allowing some “bad” moves

    But gradually decrease their step size and frequency
- Analogy with annealing:
  - At fixed temperature T, state occupation probability reaches

    Boltzman distribution $p(x) = \alpha e^{-E(x)/kT}$
  - T decreased slowly enough → always reach the best state
  - Devised by Metropolis et *al.*, 1953, for physical process modeling
  - Widely used in VLSI layout, airline scheduling, etc.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE − current.VALUE
    if ΔE < 0 then current ← next
    else current ← next only with probability e^(−ΔE/T)
```

- A random move is picked instead of the best move
- If the move improves the situation, it is always accepted
- Otherwise, the move is accepted with probability $e^{-\Delta E/T}$

- ΔE : the amount by which the evaluation is worsened
  - ○ The acceptance probability decreases exponentially with the "badness" of the move
- T : temperature, determined by the annealing schedule (controls the randomness)
  - ○ Bad moves are more likely at the start when T is high
  - ○ They become less likely as T decreases

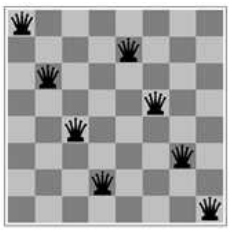
- $T \rightarrow 0$: simple hill-climbing (first-choice hill-climbing)
- If the annealing schedule lowers T slowly enough, a global optimum will be found with probability approaching 1
- The initial temperature is often heuristically set to a value so that the probability of accepting bad moves is 0.5

Genetic Algorithm

- Starts with a population of individuals
  - Each individual (state) is represented as a string over a finite alphabet (called chromosome)—most commonly, a string of 0s and 1s
- Each individual is rated by the fitness function
  - An individual is selected for reproduction by the probability proportional to the fitness score

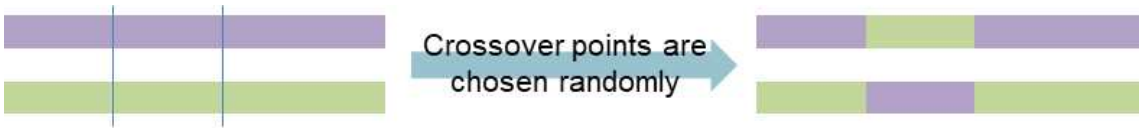
1	3	5	7	2	4	6	8

Column-by-column integer representation



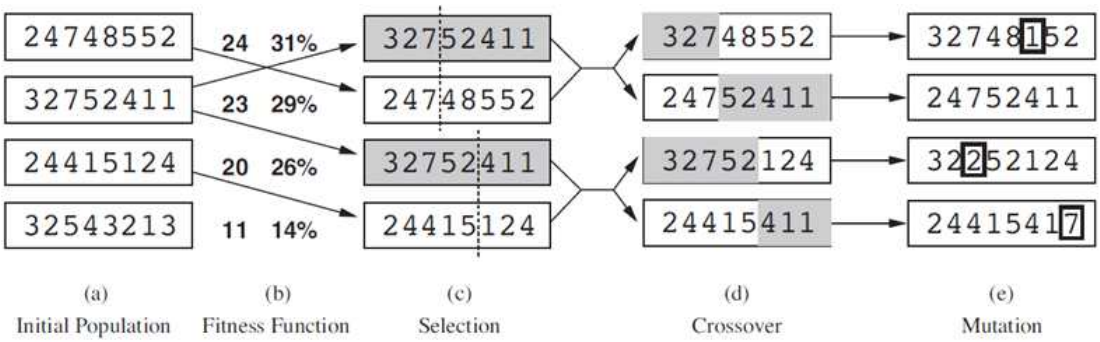
Local search algorithms do not use any problem-specific heuristics
Simulated annealing and GA use meta-level heuristic → metaheuristic algorithms

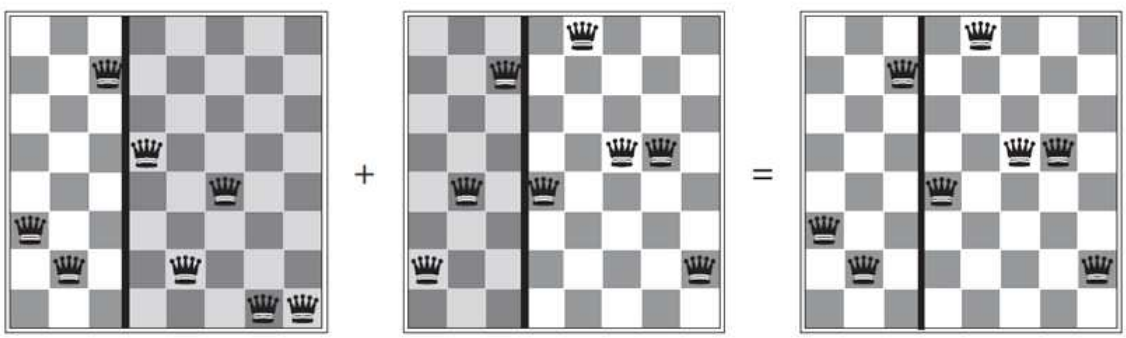
- Selected pair are mated by a crossover
  - Crossover frequently takes large steps in the state space early in the search process when the population is quite diverse, and smaller steps later on when most individuals are quite similar



- Each locus is subject to random mutation with a small independent probability
- Advantage of GA comes from crossover:
  - Is able to combine large blocks of letters that have evolved independently to perform useful functions

➡ Raises the level of granularity at which the search operates

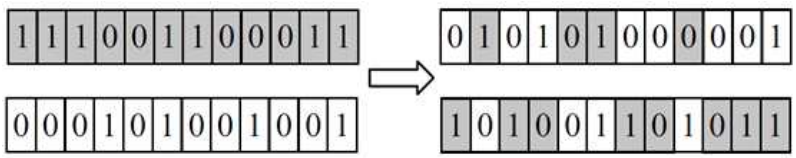




```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      (x, y) ← SELECT-PARENTS(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

- Parent selection by binary tournament:
  1. Randomly select 2 individuals with replacement
  2. Select the one with the best fitness as the winner

     (ties are broken randomly)
- Uniform crossover:
  - Each gene is chosen from either parent stochastically by flipping coin at each locus



  - If the probability of head $p = 0.5$, the average number of crossover points is $l/2$, where l is the length of the chromosome
  - $p = 0.2$ is a popular choice

- Crossover after parent selection is done according the probability called crossover rate
  - Crossover rate close to 1 is popular
- Bit-flip mutation for binary representation:
  - Each bit is flipped with a small mutation probability called mutation rate
  - Mutation rate of $1/l$ is popular

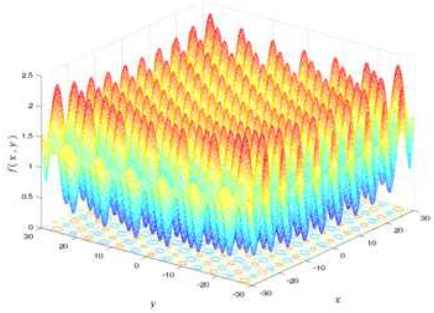
Example: Binary encoding for numerical optimization

$$\min f(x,y) = \frac{x^2+y^2}{4000} - \cos(x)\cos\left(\frac{y}{\sqrt{2}}\right) + 1 \quad (-30 \le x, y \le 30)$$

- Assuming a 10-bit binary encoding, the code shown below can be decoded as

$$-30 + (30 - (-30)) \times \frac{1}{2^{10}}(2^8 + 2^4 + 2^0)$$
$$= -14.004$$

0	1	0	0	0	1	0	0	0	1



2-D Griewank function

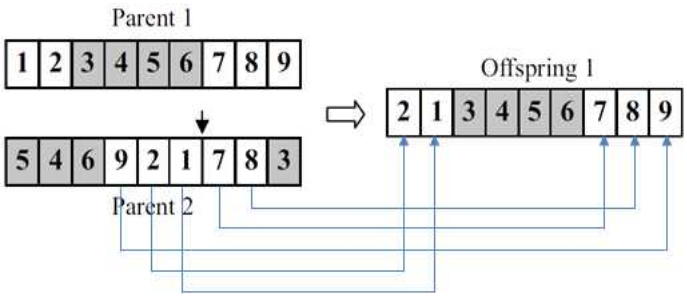
- Genetic operators for permutation code (e.g., for TSP):
  - Simple inversion mutation:

1	2	3	4	5	6	7	8	9

⇨

1	2	3	6	5	4	7	8	9

- Ordered crossover (OX):



- In the case of permutation code
  - Crossover rate is the probability of whether or not to perform the ordered crossover
  - Similarly, mutation rate is the probability of whether or not to perform the inversion