Python Examples

Chapter 1: Python Basic

Last Updated Date: **January 31, 2023**

As you can see from the example above, the code is copied from my IDE (PyCharm) verbatim. You are encouraged to copy and paste the code to your IDE. Feel free to make changes to the code and test it. I expect it to work without any editing.

We will show several sample programs that probably don't need much explanation. Each example introduces a new syntax. Be sure to know how to use these statements.

Program 1: This is the world-famous hello world program. We are allowed to use single or double quotes, but they must match.

01-hello world.py

```
1  # Name:  Stephen Huang
2  # Hello world program
3  #
4  print("Hello World!")    # In-line comment.
5  print('Hello World!')    # This works too.
```

Program 2: This example shows how we can use variables in a program. The following example uses seven variables. We also show the conventions people use to name variables.

11-input.py

```
1  # This demonstrates how to use print(), input() and a variable
2  #
3  name = input("Enter your name: ")
4  print("Good Morning,", name)
5
6  # print() can print multiple values separated by commas.
7  # Try to print the name before "Good Morning"
```

Program 3: This example shows how we can use variables in a program. The following example uses seven variables. We also show the conventions people use to name variables.

01 12-variables.py

```
1  # Variable Names
2  x = 10                  # integer
3  X = 20                  # not a good idea to have both x and X
4  x123 = 30
5  studentname = "John"    # difficult to read
6  student_name = "Bob"    # use _ to separate words
7  StudentName = "Ava"     # use upper case to separate
8  studentName = 'Betty'   # single quotes
9  print("We are using 7 different variables.")
```

Program 4.

01 13-variable print.py

```
1  def next():
2      input("> ")
3
4  print("Good Afternoon.")
5  next()
6
7  name = "John"
8  print("Good Afternoon.", name)
9  next()
10
11 greeting = "Good Morning,"
12 print(greeting, name)
13 next()
14
15 print(greeting, name, ".")    # Extra space(s)
16 next()
17
18 print(greeting, name + ".")    # this is a quick fix
19
20 # the most commonly used value are set as the default value for
21 # the parameter.  We will get to using parameters sep and end
22 # later.
```

Program 5: This example reminds us that there are certain words that we should not use as variable names. These are keywords or reserved words which are needed for the syntax. Execute the program and see the message generated. A syntax error will cause the program to terminate even if there are correct statements following the syntax error. As a beginner, you will encounter many syntax errors. The IDE will provide warnings for the error.

```
01  14-variable syntax.py
1   # This example uses a reserved word as a variable name.
2   # Syntax Error
3   finally = 99
4   print(finally)
5
6   print(True)   # Nothing wrong with the syntax here
7   # Why is nothing printed?  Before or After
8   # the syntax error.
```

```
"C:\Users\...\python.exe" "C:/Users/…/14-variables-syntax.py"
  File "C:\Users\...\14-variables-syntax.py", line 3

    finally = 99
    ^^^^^^^
SyntaxError: invalid syntax
```

Note that the error message

- pointed out the location of the error, the file name plus the line number,
- print the statement in error, and
- describe the error.

Fortunately, there are not too many such reserved words in Python. You can find a complete list online or in a textbook.

Program 6: Boolean values and relational operators.

```
01 | 15-Boolean values.py
 1 | def next():
 2 |     input("> ")
 3 |
 4 | x = 5
 5 | y = 6
 6 | print(x, y)
 7 | next()
 8 |
 9 | print(x == 5)
10 | next()
11 | print(x != 5)
12 | next()
13 | print(x < 5)
14 | next()
15 | print(x <= y)
16 | next()
17 | print(x > y)
18 | next()
19 | print(x >= y-3)
```

Program 7: Example of primitive types.

```
01 | 21-primitive.py
 1 | # We have seen integers and strings in action, now
 2 | # the other three types.
 3 | # It's better to print a label for each value that you are printing
 4 | #
 5 | def next():
 6 |     input("> ")
 7 |
 8 | a = 12345
 9 | b = 3.14
10 | c = True
11 | d = "Hello"
12 | e = 8+6j
13 | print ("a = ", a, "  b = ", b, "  c = ", c, "  d = ", d, "  e = ", e)
14 | next()
15 |
16 | a = 12345
17 | b = 2/3
18 | c = False
19 | d = '\tHello'
20 | e = 8-6j
21 | print ("a = ", a, "  b = ", b, "  c = ", c, "  d = ", d, "  e = ", e)
```

Program 8:

01 22-boolean.py

```
 1 # True or False
 2 # Boolean values
 3 #
 4 def next():
 5     input("> ")
 6
 7 print("False", False)
 8 print("True", True)
 9 next()
10
11 print("None", None==True)
12 print("None", None==False)
13 next()
14
15 value = 0
16 print(value, "is interpreted as", value==True)
17 next()
18
19 value = 1
20 print(value, "is interpreted as", value==True)
21 next()
22
23 print("Empty Sequence", []==True)
24 next()
25
26 print(""==True)
27 next()
28
29 print(123.45==True)
30 print("test"==True)
31 print("0"==True)
```

Program 9: Casting one type into another.

```
01 | 23a-casting.py
 1 | def next():
 2 |     input("Next> ")
 3 |
 4 | print(f"1. {True}, {int(True)}, {float(True)}")
 5 | print(f"1. {False}, {int(False)}, {float(False)}")
 6 | next()
 7 |
 8 | print(f"2. {bool(-1.5)}, {bool(0)}, {bool(2)}")
 9 | next()
10 |
11 | print(f"3. {2}, {bool(2)}, {int(bool(2))}")
12 | next()
13 |
14 | print(f"4. {True}, {int(True)}, {bool(int(True))}")
15 | next()
16 |
17 | print(f"5. {99}, {bool(99)}, {int(bool(99))}")
18 | next()
19 |
20 | print(f"6. {3.14159}, {int(3.14159)}, {float(int(3.14159))}")
21 | print(f"6. {3.99999}, {int(3.9999)}")
22 | print(f"6. {3}, {float(3)}")
```

Program 10: Casting one type into another.

```
01 | 23b-casting.py
 1 | def next():
 2 |     input("> ")
 3 |
 4 | age = 20
 5 | weight = 123.95
 6 | id = '123-45-6789'
 7 | okay = True
 8 | print(f"1: age = {age}, {float(age)}, [{str(age)}]")
 9 | next()
10 |
11 | print(f"2: weight = {weight}, {int(weight)}, [{str(weight)}]")
12 | next()
13 |
14 | print(f"3. id = [{id}] XXX XXX")
15 | next()
16 |
17 | print(f"4: okay = {okay}, {int(okay)}, {float(okay)},
18 | [{str(okay)}]")
19 | next()
20 |
21 | print(f"5: age = {age}, {bool(age)}, [{str(age)}]")
22 | next()
23 |
24 | id2 = '12345'           # Try 'test', '3e5'
25 | print(f"6: id2 = [{id2}], {float(id2)}, {int(id2)}")
```

Program 11: This example demonstrates two functions. The id() function gives you the memory location of the value stored in the variable. The type() function gives you the variable type (int, str, etc.).

01 24-dynamic.py

```
 1 def next():
 2     input("Next > ")
 3
 4 i = 0
 5 print("1. i, ID =", id(i), "Type =", type(i), "Value =", i)
 6 next()
 7
 8 i = i+1
 9 print("2. i, ID =", id(i), "Type =", type(i), "Value =", i)
10 next()
11
12 j, k = 0, 1
13 print("3. j, ID =", id(j), "Type =", type(j), "Value =", j)
14 print("3. k, ID =", id(k), "Type =", type(k), "Value =", k)
15 next()
16
17 j = k = 1234
18 print("4. j, ID =", id(j), "Type =", type(j), "Value =", j)
19 print("4. k, ID =", id(k), "Type =", type(k), "Value =", k)
20 next()
21
22 i = "Hello"
23 j = "World"
24 k = "Hello"
25 print("5. i, ID =", id(i), "Type =", type(i), "Value =", i)
26 print("5. j, ID =", id(j), "Type =", type(j), "Value =", j)
27 print("5. k, ID =", id(k), "Type =", type(k), "Value =", k)
```

Program 12: This shows the difference between Casting and Coercion: Casting is explicit, and Coercion is implicit. The two operands must be of the same type. A computer cannot add an int and a float together.

```
01 | 25-coerce.py
 1 | def next():
 2 |     input("Next> ")
 3 |
 4 | name = "John"
 5 | age = 24
 6 | print(f"1. name, ID = {id(name)}, Type = {type(name)}, "
 7 |       f"Value = {name}, length = {len(name)}")
 8 | next()
 9 |
10 | print(f"2. age,  ID = {id(age)}, Type = {type(age)}, "
11 |       f"Value = {age}")
12 | next()
13 |
14 | x = name + " " +str(age)
15 | print(f"3. sum,  ID = {id(x)}, Type = {type(x)}, "
16 |       f"Value = [{x}], length = {len(x)}")
17 | next()
18 |
19 | a = 20
20 | b = 3.59
21 | sum = a + b
22 | print(f"4. a,  ID = {id(a)}, Type = {type(a)}, Value = {a}")
23 | next()
24 |
25 | print(f"5. b,  ID = {id(b)}, Type = {type(b)}, Value = {b}")
26 | next()
27 |
28 | print(f"6. sum,  ID = {id(sum)}, Type = {type(sum)}, Value = {sum}")
29 | next()
30 |
31 | sum = float(a) + b
32 | print(f"7. sum,  ID = {id(sum)}, Type = {type(sum)}, Value = {sum}")
```

Program 13: This example computes the volume of a cylinder using radius and height.

01 31-exp-1.py

```
import math
def next():
    input("Next > ")

radius, height = 10, 5
volume = math.pi * radius**2 * height
print(f"1. Radius = {radius}, Height = {height}, Volume =
{volume}.")
next()

radius, height = 8, 4
print(f'2. Radius = {radius}, Height = {height}, '
      f'Volume = {math.pi * radius**2 * height}.')
next()

radius, height = 8, 4
print(f'3. Radius = {radius}, Height = {height}, '
      f'Volume = {math.pi * radius**2 * height:10.3f}.')
```

Program 14: f-string and formatting.

01 32-exp-2.py

```
import math
def next():
    input("Next > ")

print(f'1. math.pi is {math.pi}')
next()

print(f'2. math.pi is [{math.pi:5.2f}]')
next()

r = 10
print(f'3. Area (pi*r*r)  {math.pi*r*r: 10.4f}')
```

Program 15: Python allows you to chain several function calls together if an inner function's output is the outer function's input.

```
01 | 33-exp-3.py
 1 | x = input("Enter a number: ")
 2 | x = int(x)
 3 | print(f"1. x = {x}")
 4 |
 5 | num = int(input("Give me an int: "))
 6 | print(f"2. x*num = {x*num}")
 7 |
 8 | num = float(input("Give me a float: "))
 9 | print(f"3. x*num = {x*num}")
10 |
11 | num = float(input("Give me a float: "))*x
12 | print(f"4. num = {num}, num changed")
13 |
14 | # Everything together, is it better?
15 | print(f"5. result = {float(input('Give me a float: ')) * x}")
```

Program 16: If we want to assign the same value to multiple variables, we can do it in one assignment. This situation happens a lot during the initialization of variables.

```
01 | 34-multi-assign.py
 1 | def next():
 2 |     input("Next > ")
 3 |
 4 | # Assigning the same value to multiple variables
 5 | a = b = c = 99
 6 | print(a, b, c)
 7 | next()
 8 |
 9 | a = b = 0
10 | print(a, b, c)
```

Program 17: Multiple examples of parallel assignments are shown here. Study the swap carefully. It is handy for swapping two variables.

```
01 | 35-parallel-assign.py
```

```
 1  def next():
 2      input("Next> ")
 3
 4  # Assigning parallel values
 5  x, y = 1, 99
 6  print("Case 0: ", x, y)
 7  next()
 8
 9  # Swap -- simultaneously or in parallel
10  x, y = y, x
11  print("Case 1:", x, y)
12  next()
13
14  # 2 Statements swap -- sequentially, wrong
15  x, y = 1, 99
16  x = y
17  y = x
18  print("Case 2:", x, y)
19  next()
20
21  # 3 Statements swap -- sequentially, right
22  x, y = 1, 99
23  temp = x
24  x = y
25  y = temp
26  print("Case 3:", x, y)
27  next()
28
29  # Same number of items on both sides of =
30  x, y, z = 1, 2, 3    # try 1, 2, 3, 4
31  print("Case 4:", x, y, z)
32  next()
33
34  x = 'test'
35  y = len(x)
36  print("Case 5:", x, y)
37  next()
38
39  x, y = 'testing', len(x)
40  print("Case 6:", x, y)
41  # Why?
```

Program 18: Examples of arithmetic operators.

01 36-unpacking.py

```
 1 def next():
 2     input("Next> ")
 3
 4 fruits = ['apple', 'banana', 'cherry', 'pineapple']
 5 print(fruits)
 6 var1, var2, var3, var4 = fruits
 7 print(f"1.  {var1}, {var2}, {var3}, {var4}")
 8 next()
 9
10 [var1, var2, var3, var4] = fruits
11 print(f"2.  {var1}, {var2}, {var3}, {var4}")
12 next()
13
14 *varx, var3, var4 = fruits
15 print(f"3.  {varx}, {var3}, {var4}")
16 next()
17
18 var1, *varx, var4 = fruits
19 print(f"4.  {var1}, {varx}, {var4}")
```

Program 19: Examples of arithmetic operators.

01 40-arithmetic-op.py

```
 1 # Keywords: arithmetic operators
 2 #
 3 def next():
 4     input("Next > ")
 5
 6 a, b = 20, 7
 7 print(a+b)
 8 next()
 9 print(a-b)
10 next()
11 print(a*b)
12 next()
13 print(a/b)
14 next()
15 print(a//b)
16 next()
17 print(a%b)
18 next()
19 print(a**b)
```

Program 20: This example shows that precedence does matter. Try all commonly used operators on your own. Learn what the default precedence (depends on the operators) is.

41-precedence.py

```
def next():
    input("Next > ")

i, j, k = 2, 5, 81
print(i, j, k)
next()

print(k/j/i, k/(j/i))
next()

print(k//j//i, k//(j//i))
next()

print(3**2**3)
next()

print((3**2)**3, 3**(2**3))
```

Program 21: Practice the relational operators.

42-relational.py

```
# Keywords: relational operators
#
a, b = 20, 7
print(a>b)
print(a<b)
print(a==b)
print(a!=b)
print(a>=b)
print(a<=b)

# Float values are not precise.  Avoid equal comparison.
```

Program 22: Examples of other ways of using an assignment statement and **augmented operator**. Augmented operators are binary operators in an expression. You may think of these augmented operators as shorthand for an assignment. There is a big difference, though. There is a "side effect" that the evaluation also changed the variable on the left. Test all operators that we have seen before.

```
01 | 43-augmented op.py
 1 | def next():
 2 |     input("Next > ")
 3 |
 4 | var = 'test'
 5 | print("*** Case 1: simple assignment: ", var)
 6 | next()
 7 |
 8 | a = b = "Go Coogs!"
 9 | var = a+b
10 | print("*** Case 2: multiple assignment, simple assignment: ", var)
11 | next()
12 |
13 | a, b = 2, 3
14 | var = a**2 + b**2
15 | print("*** Case 3: parallel assignment, simple assignment: ", var)
16 | next()
17 |
18 | a += b+99
19 | b *= 5
20 | print("*** Case 4: augmented assignment: ", a, b)
21 | next()
22 |
23 | a -= 1
24 | b **= 2
25 | print("*** Case 5: augmented assignment: ", a, b)
```

Program 23: This example demonstrates how to use the assignment operators. That's right, these symbols (such as *=) are operators, not assignments. The operators make a change to the variables on the left-hand side. Assignment operators are performing assignments, so you cannot use them as part of an expression. The following statement won't work: "`print(a+=2)`". Try it.

```
01 | 44-assignment-op.py
 1 | def next():
 2 |     input("Next > ")
 3 |
 4 | a, b = 20, 7
 5 | a = b
 6 | print("Case 1: ", a)
 7 | next()
 8 |
 9 | a, b = 20, 7
10 | a += b      # a = a + b
11 | print("Case 2: ", a)
12 | next()
13 |
14 | a, b = 20, 7
15 | a -= b
16 | print("Case 3: ", a)
17 | next()
18 |
19 | a, b = 20, 7
20 | a *= b
21 | print("Case 4: ", a)
22 | next()
23 |
24 | a, b = 20, 7
25 | a/=b
26 | print("Case 5: ", a)
27 | next()
28 |
29 | a, b = 20, 7
30 | a//=b
31 | print("Case 6: ", a)
32 | next()
33 |
34 | a, b = 20, 7
35 | a %= b
36 | print("Case 7: ", a)
37 | next()
38 |
39 | a, b = 20, 7
40 | a **= b
41 | print("Case 8: ", a)
```

Program 24: Practice logical operators. NOT > AND > OR.

01 45-logical.py

```
# Keywords: logical operators
#
def next():
    input("Next > ")

a, b, c = True, False, True
print("Case 1: ", a, b)
next()

print("Case 2: ", a and b)
next()

print("Case 3: ", a or b)
next()

print("Case 4: ", not a)
next()

print("Case 5: ", a and not b or c)
next()

print("Case 6: ", (a and (not b)) or c)
next()

print("Case 7: ", a and not (b or c))
```

Program 25: Identity function.

```
01 | 46-is identity.py
 1 | def next():
 2 |     input("Next > ")
 3 |
 4 | a, b, c = 7, 100, 7
 5 | print(f"a  {id(a)}  {a}")
 6 | print(f"b  {id(b)}  {b}")
 7 | print(f"c  {id(c)}  {c}")
 8 | next()
 9 |
10 | print(a==b, a is b)
11 | next()
12 |
13 | print(a==c, a is c)
14 | next()
15 |
16 | a = b = [1, 2, 3]
17 | c = [1]+[2, 3]
18 |
19 | print(a==b, a is b, id(a), id(b), a, b)
20 | next()
21 | print(a==c, a is c, id(a), id(c), a, c)
```

Program 26: Membership testing.

```
01 | 47-membership.py
 1 | # Keywords: membership operators
 2 | #
 3 | print(6 in [1, 2, 3, 4, 5])
 4 | print('u' in "quit")
 5 | print(6 not in [1, 2, 3, 4, 5])
 6 | print('u' not in "quit")
 7 | print(not 'q' in "quit")
```

Program 27: Different ways to break a statement into multiple lines. The print() function allows us to print multiple values. Pretend the examples we used here are very long.

```
01 | 51-join-lines.py
 1 | def next():
 2 |     input("Next > ")
 3 |
 4 | print("1. In the face of ambiguity,"  # why does it work?
 5 |       "refuse the temptation to guess.")
 6 | next()
 7 |
 8 | print("2. In the face of ambiguity," + # concatenation as is
 9 |       "refuse the temptation to guess.")
10 | next()
11 |
12 | print("3. In the face of ambiguity,",  # notice the difference?
13 |       "refuse the temptation to guess.")
14 | next()
15 |
16 | print("4. In the face of ambiguity, \
17 |         refuse the temptation to guess.")  # nothing after \
18 | next()
19 |
20 | print("5. In the face of ambiguity, \
21 | refuse the temptation to guess.")  # which one is better?
```

Program 28: We typically put one statement per line in Python. This program shows it is possible to put multiple statements in one line, which is a BAD practice. Please don't do it. We also show how to place a statement across multiple lines, which is sometimes necessary.

01 52-statement-line.py

```
# keywords: Logical vs. Physical lines
# Implicit vs. Explicit Join
def next():
    input("\nNext > ")

print("\tCase 1:")
# Multiple statements in a line
print("Hello "); print("world 1")    # bad
print("Hello "); print("world 2");   # really bad
next()

# A statement across multiple lines
print("\tCase 2:")
print("Hello ", end="*")     # rewrite the end-of-line character
print("world 3")
next()

# Breaking a quote pair,
# Joining two physical lines
print("\tCase 3:")
print('Hello \
world \
4')
next()

# Breaking a list
print("\tCase 4:")
print([10, 20, 30, \
       40, 50])
print([10, 20, 30,        # it is okay without \, preferred, why?
       40, 50])
# This does not work on strings, try to remove \ in Case 3
```

Program 29: Introducing the None value.

```
01 | 53-none.py
 1 | # An empty string is a string with zero characters,
 2 | # it is a legitimate string.
 3 |
 4 | a = None
 5 | b = ''
 6 | c = 'T e s t'
 7 | print(f"[{a}], [{b}], [{c}]")
 8 |
 9 | a = 3.14
10 | b = 5
11 | c = None
12 | print(f"[{a}], [{b}], [{c}]")
```

Program 30: Escape character and eoln.

```
01 | 61-eoln.py
 1 | # Keywords: Escape characters, \n, eoln
 2 | #
 3 | def next():
 4 |     input("Next > ")
 5 |
 6 | myString = "\n"
 7 | print("1. length =", len(myString), "String = ["+myString+"]")
 8 | next()
 9 |
10 | myString = "2. first line\nsecond line.\n"
11 | print(myString, len(myString))
12 | next()
13 |
14 | myString = "3. first line[\n]second line.\n"
15 | print( myString, len(myString))
16 | next()
17 |
18 | myString = "\n4. first line\n\tsecond line."
19 | print(myString, len(myString))
```

Program 31: Examples of escape characters.

```
01  62-escape.py
 1  def next():
 2      input("\nNext > ")
 3
 4  print("hello world")
 5  print("\thello world")
 6  print("\t\thello world")
 7  next()
 8
 9  # Likely to use
10  print("Tab: [\t]")
11  print("Eoln: [\n]")
12  print("Quote: [\']")
13  print('Quote: [\"]')
14  print("Backslash: [\\]")
15  next()
16
17  # Not likely to use
18  print("Backspace: [\b]")
19  print("Carriage Return: [\r]")
```

Program 32: Single and Double quotes.

```
01  63-quote.py
 1  # Keywords: quotes
 2  # what is the benefit of having two quotes?
 3  def next():
 4      input("Next > ")
 5  #
 6  print('single \' double " inside a single quote.')
 7  next()
 8  print("single ' double \" inside a double quote.")
 9  next()
10  print("single \' double \" inside a double quote.")
11  next()
12  print('\\\\')
```

Program 33: Concatenation and repetition operators.

01 64-conca-rep.py

```
# Keywords: concatenation, repetition operators
def next():
    input("Next > ")

print('star '+'wars')
next()

print('star'+'wars')
next()

print('Tora! '*3)
next()

print("10"*4)
next()

print("4"*10)
```

Program 34: Named parameters for print.

01 65-sep-end.py

```
# Keywords: end sep "named" parameters
# Overriding the default
# One more reason to use f-string
def next():
    input("Next > ")

person = input("Enter your name: ")
print("1. Hello",    person)
next()
print("2. Hello"   + person)
next()
print("3. Hello "  + person)
next()
print("4.  Hello", person, '!')
next()
print("5. Hello", person, sep='', end='!')
next()
print("6. Hello", person, sep='_', end='***')
next()
print("7. Hello ", person, '!', sep='')
next()
print(f"8. Hello, {person}!")
```

Program 35: End-of-Line character.

```
01 66-eoln.py
 1 quotation = "1. Well-written code \
 2 is its own best documentation."
 3 print(quotation)
 4
 5 quotation = "2. Well-written code " + \
 6             "is its own best documentation."
 7 print(quotation)
 8
 9 quotation = "3. Well-written code " \
10             + "is its own best documentation."
11 print(quotation)
12
13 quotation = "4. Well-written code " \
14             "is its own best documentation."
15 print(quotation)
16
17 quotation = "5. Well-written code \
18 is its own best documentation."
19 print(quotation)
```

Program 36: All inputs are in string form, even if it looks like you entered a number. The string in the input function is there to prompt the user to enter something. Here we also print out the type of the value received. The function type() comes with the language.

```
01 71-input.py
 1 num = input("How old are you? ")
 2 print(num, type(num))
```

Program 37: Input follow by casting.

01 72-input-cast.py

```
def next():
    input("Next > ")

num = input("How old are you? ")
print("Case 1:", num, type(num))
next()

num = int(num)
print("Case 2:", num, type(num))
next()

num += 1
print("Case 3:", num, type(num))
next()

num = int(input("How old are you? "))
print("Case 4:", num, type(num))
next()

# If you don't need to save the value, even shorter
num = print("Case 5: You are ", int(input("How old are you? ")), '
years old.', sep='')
print('Not recommended.')
```

Program 38: We can use int() or float() to cast the string into a corresponding numerical value. If you want to let the user enter either int or float, you should use eval(), which evaluates the string and produce a number depending on what was entered.

01 73-input-eval.py

```
num = int(input("How old are you? "))
print("Case 1:", num, type(num))    # try 55, also 55.6

num = float(input("How old are you? "))
print("Case 2:", num, type(num))    # try 66.6, also 66

# what if you are not sure, int or float
num = eval(input("Enter a number: "))
print("Case 3:", num, type(num))   # try 45

num = eval(input("Enter a number: "))
print("Case 4:", num, type(num))   # try 44.55

num = eval(input("Enter a number: "))
print("Case 5:", num, type(num))   # try 'Two' or '2'
```

Program 39: This example shows how we import a module. We first import the module. Then we are allowed to use everything inside the module. "math.pi" means "pi" inside the module "math". In the example, we would like to use the pi value, which is stored in a module called "math".

A second purpose of the example is to show how we can format a floating-point value. The first number in "10.5f" specifies the width of the output, and the second number specifies the number of digits after the decimal point or precision. It turns out that the format width is just a **'suggestion'** to the computer. If there is insufficient space to print the number, it automatically increases the width (see Case 5).

```
01 | 81-obj-import.py
 1 | def next():
 2 |     input("Next > ")
 3 |
 4 | import math
 5 |
 6 | print(f"1. math.pi is {math.pi}")
 7 | next()
 8 |
 9 | print(f'2.      math.pi is {math.pi:10.5f}')
10 | next()
11 |
12 | print(f'3. 1000*math.pi is {1000*math.pi:10.2f}')
13 | next()
14 |
15 | #How do we print just one space before the number?"
16 | print(f'4. math.pi is {math.pi:7.5f}')
17 | next()
18 |
19 | print(f'5. math.pi*1000 is {1000*math.pi:4.2f}')
20 | next()
```

Program 40: "Pass" is a statement that does nothing. Why do we need such a statement? It turns out that even though it does nothing, it is a statement. Sometimes we must put a statement in place. Ignore the "`if … elif … else`" syntax now.

```
01 | 82-pass.py
 1 | num = int(input("Enter an integer: "))
 2 |
 3 | if num==0:
 4 |     print("Are you sure?") # test this now
 5 | elif num>0:
 6 |     pass # handle this part later
 7 | else:
 8 |     pass # handle this part later
```

Program 41: More examples of using modules and objects. We are not going to discuss objects much in this course. Two new features of import are introduced here. (a) You can import part of a module. (b) You can give a module a short name. The second half of the code measures the amount of time it takes to run a section of code.

```
01 | 83-module.py
 1 | def next():
 2 |     input("Next> ")
 3 |
 4 | from datetime import datetime
 5 | import time
 6 | import calendar as cal
 7 |
 8 | c = cal.TextCalendar(cal.SUNDAY)
 9 | print("Set the month calendar to start a week with SUNDAY")
10 | c.prmonth(2023,1)
11 | next()
12 |
13 | print(f"2. Current Date/Time: {datetime.now()}")
14 | next()
15 |
16 | print(f"3. Before: {time.time():10.6f}")
17 | next()
18 |
19 | for i in range(100000):
20 |     pass
21 |
22 | print(f"4.  After: {time.time():10.6f}")
```