UE18CS101: INTRODUCTION TO COMPUTING USING

python™

Department of Computer Science and Engineering
PES UNIVERSITY

Lecture 5 and 6

Output function, Variables, types and id, Input function

Basic Input

The programs that we will write request and get information from the user. In Python, the **input function** is used for this purpose,

```
name = input('What is your name?: ')
```

Characters within quotes are called strings. This particular use of a string, for requesting input from the user, is called a **prompt**.

The input function displays the string on the screen to prompt the user for input,

```
What is your name?: Charles
```

The underline is used here to indicate the user's input.

Basic Output

The print function is used to display information on the screen in Python.
This may be used to display a message (string),

```
>>> print('Welcome to My First Program!')
Welcome to My First Program!
```

or used to output the value of a variable,

```
>>> n = 10
>>> print(n)
10
```

PES UNIVERSITY

Can also display a combination of strings and variables,

```
>>> name = input('What is your name?: ')
What is your name?: Charles

>>> print('Hello', name)
Hello Charles
```

Note that a comma is used to separate the individual items being printed, which causes a space to appear between each when displayed. Thus, the output of the print function in this case is

```
Hello Charles
```

and not

```
HelloCharles
```

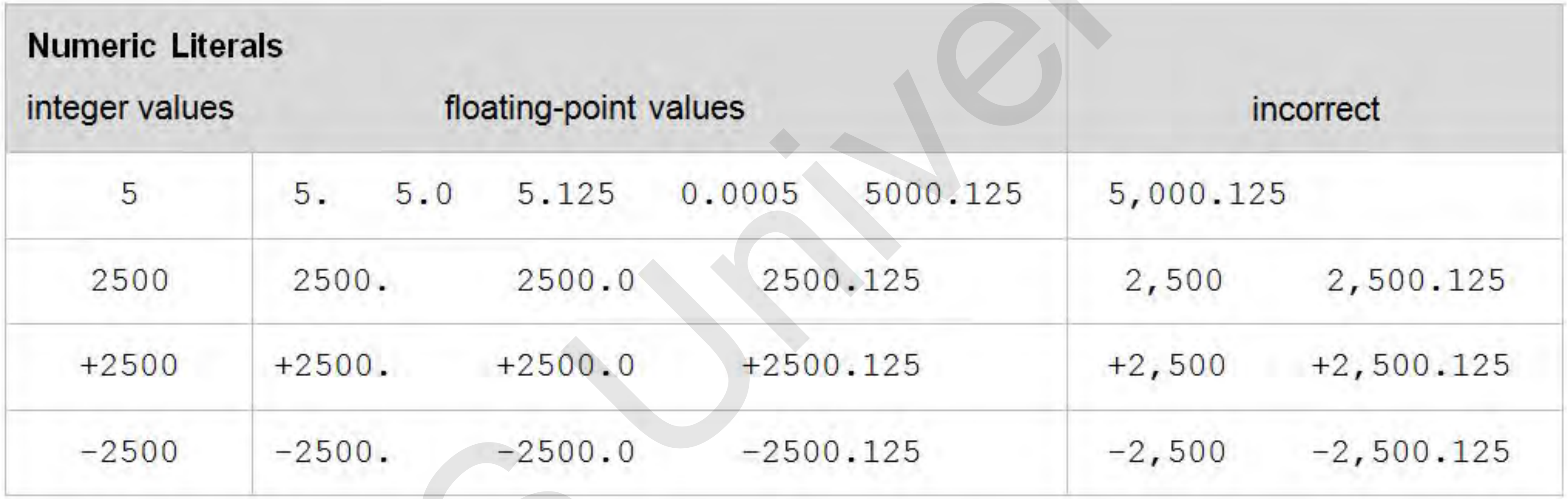
Numeric Literals

A numeric literal **is a literal containing only the digits 0–9, an optional sign character (+ or -), and a possible decimal point**. (The letter **e** is also used in exponential notation, shown next).

If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or “ float ” (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10).

Commas are never used in numeric literals .

Example Numerical Values



Numeric Literals

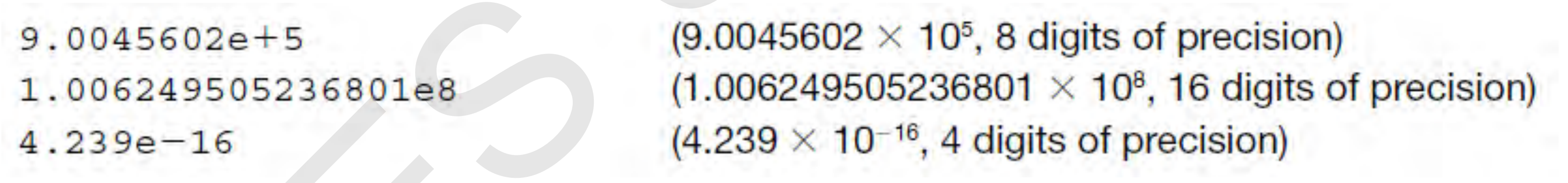
integer values	floating-point values	incorrect
5	5. 5.0 5.125 0.0005 5000.125	5,000.125
2500	2500. 2500.0 2500.125	2,500 2,500.125
+2500	+2500. +2500.0 +2500.125	+2,500 +2,500.125
-2500	-2500. -2500.0 -2500.125	-2,500 -2,500.125

Since numeric literals without a provided sign character denote positive values, an explicit positive sign is rarely used.

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating point values, however, have both a limited ***range*** and a limited ***precision***.

Python uses a double-precision standard format (IEEE 754) providing a range of 10^{-308} to 10^{308} with 16 to 17 digits of precision. To denote such a range of values, floating-points can be represented in scientific notation.



`9.0045602e+5`	(9.0045602×10^{5}, 8 digits of precision)
`1.006249505236801e8`	($1.006249505236801 \times 10^{8}$, 16 digits of precision)
`4.239e-16`	(4.239×10^{-16}, 4 digits of precision)

It is important to understand the limitations of floating-point representation. For example, the multiplication of two values may result in ***arithmetic overflow*****, a condition that occurs when a calculated result is too large in magnitude (size) to be represented**,

```
>>> 1.5e200 * 2.0e210
inf
```

This results in the special value **`inf`** (“infinity”) rather than the arithmetically correct result 3.0e410, indicating that arithmetic overflow has occurred.

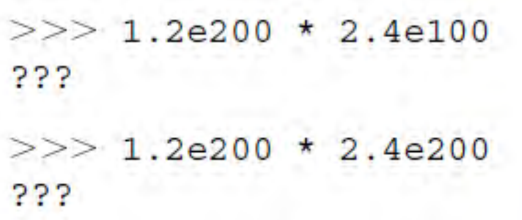
Similarly, the division of two numbers may result in ***arithmetic underflow*****, a condition that occurs when a calculated result is too small in magnitude to be represented**,

```
>>> 1.0e-300 / 1.0e100
0.0
```

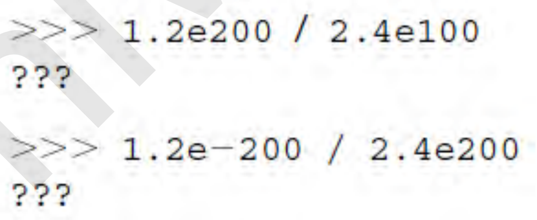
This results in 0.0 rather than the arithmetically correct result 1.0e-400, indicating that arithmetic underflow has occurred.

Let’s Try It

What do each of the following arithmetic expressions evaluate to?



```
>>> 1.2e200 * 2.4e100
???
>>> 1.2e200 * 2.4e200
???
```



```
>>> 1.2e200 / 2.4e100
???
>>> 1.2e-200 / 2.4e200
???
```

Limits of Precision in Floating-Point Representation

Arithmetic overflow and arithmetic underflow are easily detected. The loss of precision that can result in a calculated result, however, is a much more subtle issue. For example, 1/3 is equal to the infinitely repeating decimal .33333333 . . ., which also has repeating digits in base two, .010101010. . . . Since any floating-point representation necessarily contains only a finite number of digits, what is stored for many floating-point values is only an *approximation of the true value,* as can be demonstrated in Python,

```
>>> 1/3
.3333333333333333
```

Here, the repeating decimal ends after the 16th digit. Consider also the following,

```
>>> 3 * (1/3)
1.0
```

Given the value of 1/3 above as .3333333333333333, we would expect the result to be .9999999999999999, so what is happening here?

The answer is that **Python displays a rounded result to keep the number of digits displayed manageable**. However, the representation of 1/3 as .3333333333333333 remains the same, as demonstrated by the following,

```
>>> 1/3 + 1/3 + 1/3 + 1/3 + 1/3 1 + 1/3
1.9999999999999998
```

In this case we get a result that reflects the representation of 1/3 as an approximation, since the last digit is 8, and not 9. However, if we use multiplication instead, we again get the rounded value displayed,

```
>>>6 * (1/3)
2.0
```

The bottom line, therefore, is that **no matter how Python chooses to display calculated results, the value stored is limited in both the range of numbers that can be represented and the degree of precision**. For most everyday applications, this slight loss in accuracy is of no practical concern. However, in scientific computing and other applications in which precise calculations are required, this is something that the programmer must be keenly aware of.

PES UNIVERSITY

Built-in Format Function

Because floating-point values may contain an arbitrary number of decimal places, the **built-in format function can be used to produce a numeric *string* version of the value containing a specific number of decimal places,**

```
>>>  12/5                 >>> 5/7
2.4                       0.7142857142857143
>>>  format(12/5, '.2f')  >>> format(5/7, '.2f')
'2.40'                    '0.71'
```

In these examples, **format specifier '.2f'** rounds the result to two decimal places of accuracy in the string produced.

For very large (or very small) values '**e**' can be used as a format specifier.

```
>>> format(2 ** 100, '.6e')
'1.267651e+30'
```

Here, the value is formatted in scientific notation, with six decimal places of precision.

Formatted numeric string values are useful when displaying results in which only a certain number of decimal places need to be displayed:

Without use of format specifier:

```
>>> tax = 0.08
>>> print('Your cost: $', (1 + tax) * 12.99)
Your cost: $ 14.029200000000001
```

With use of format specifier:

```
>>> print('Your cost: $', format((1 + tax) * 12.99, '.2f'))
Your cost: $ 14.03
```

Finally, a comma in the format specifier adds comma separators to the result.

```
>>> format(13402.25, ',.2f')
13402.24
```

String Literals

String literals , or " strings ," represent a sequence of characters.

```
'Hello'   'Smith, John'   "Baltimore, Maryland 21210"
```

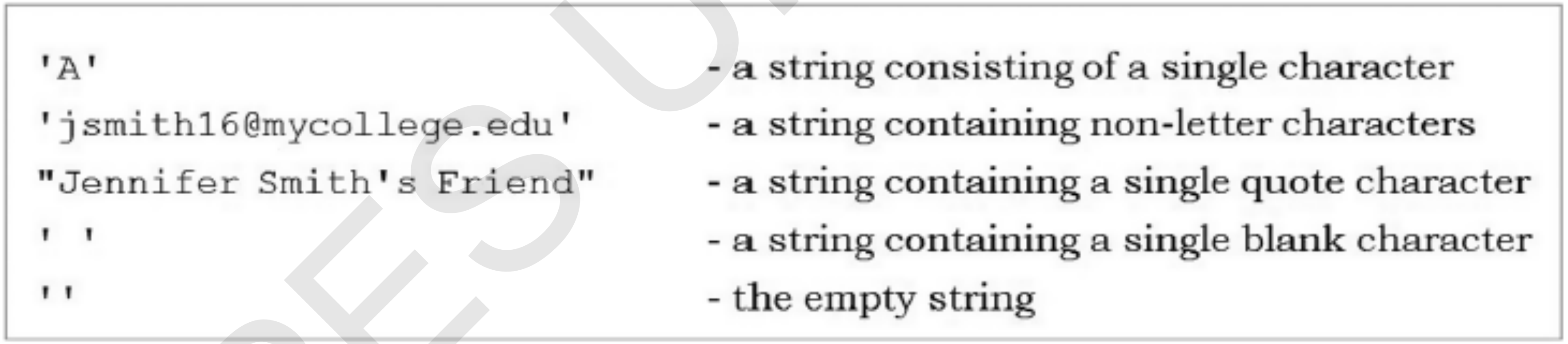
In Python, string literals may be delimited (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line (except when delimited by triple quotes, discussed later).

We have already seen the use of strings in Chapter 1 for displaying screen output,

```
>>> print('Welcome to Python!')
Welcome to Python!
```

A string may contain zero or more characters, including **letters**, **digits**, **special characters**, and **blanks**. A string consisting of only a pair of matching quotes (with nothing in between) is called the **empty string**, which is different from a string containing only blank characters. Both blank strings and the empty string have their uses, as we will see.

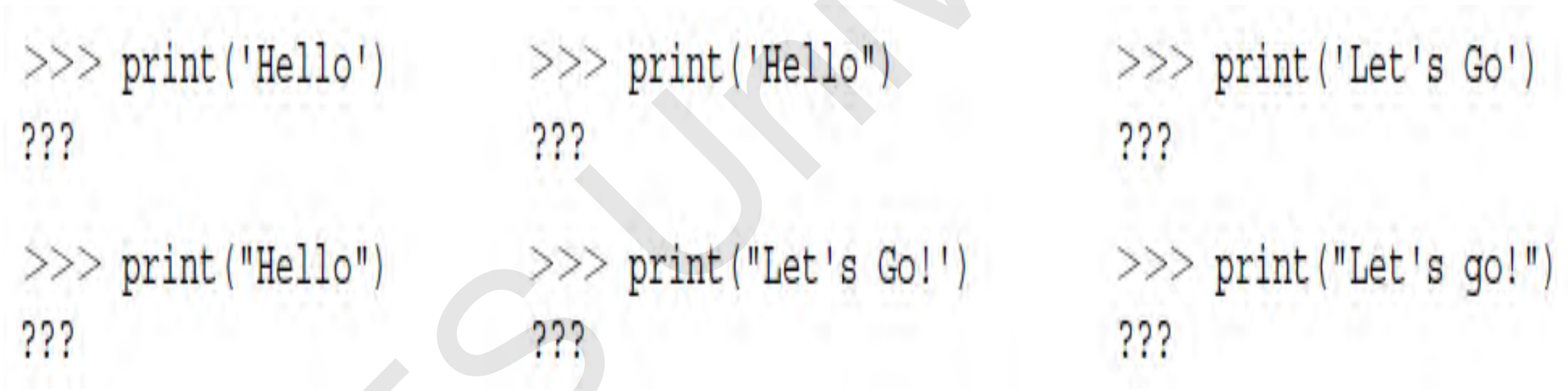
Strings may also contain quote characters as long as different quotes (single vs. double) are used to delimit the string.



`'A'`	- a string consisting of a single character
`'jsmith16@mycollege.edu'`	- a string containing non-letter characters
`"Jennifer Smith's Friend"`	- a string containing a single quote character
`' '`	- a string containing a single blank character
`''`	- the empty string

Let’s Try It

What will be displayed by each of the following?



```
>>> print('Hello')        >>> print('Hello")        >>> print('Let's Go')
???                       ???                       ???

>>> print("Hello")        >>> print("Let's Go!')    >>> print("Let's go!")
???                       ???                       ???
```

The Representation of Character Values

There needs to be a way to encode (represent) characters within a computer. Although various encoding schemes have been developed, the **Unicode encoding scheme** is intended to be a universal encoding scheme.

Unicode is actually a collection of different encoding schemes utilizing between 8 and 32 bits for each character. The default encoding in Python uses **UTF-8**, an 8-bit encoding compatible with **ASCII**, an older, still widely used encoding scheme.

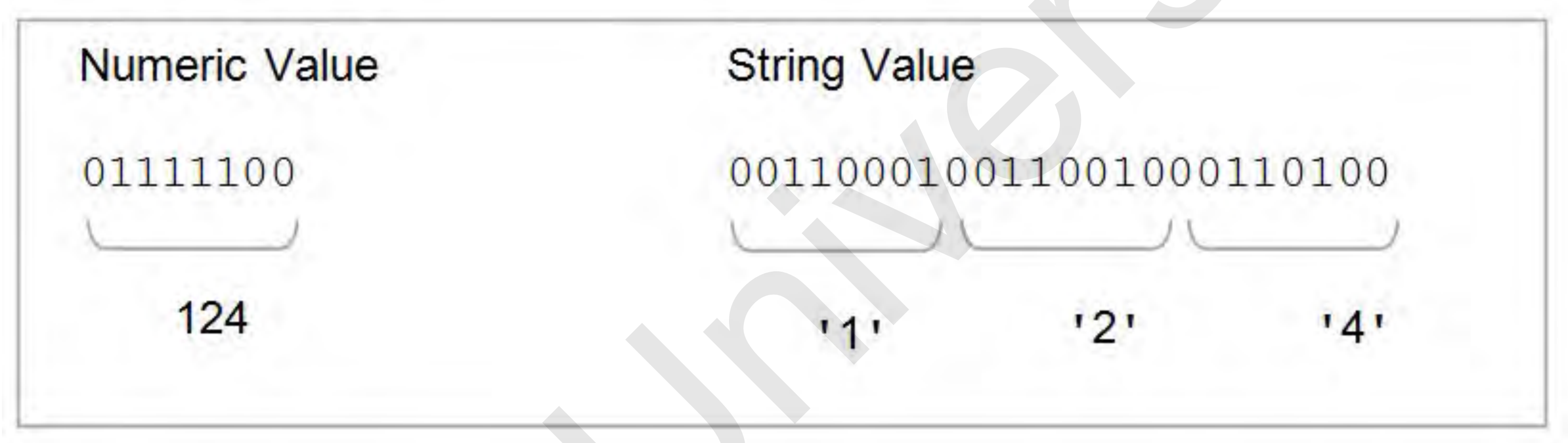
Currently, there are over 100,000 Unicode-defined characters for many of the languages around the world. Unicode is capable of defining more than four billion characters. Thus, all the world's languages, both past and present, can potentially be encoded within Unicode.

Space	00100000	32	A	01000001	65
!	00100001	33	B	01000010	66
"	00100010	34	C	01000011	67
#	00100011	35	.		
.			.		
.			Z	01011010	90
0	00110000	48	a	01100001	97
1	00110001	49	b	01100010	98
2	00110010	50	c	01100011	99
.			.		
.			.		
9	00111001	57	z	01111010	122

Partial listing of the ASCII-compatible UTF-8 encoding scheme

UTF-8 encodes characters that have an ordering with sequential numerical values. For example, 'A' is encoded as 01000001 (65), 'B' is encoded as 01000010 (66), and so on. This is also true for digit characters, ‘0’ is encoded as 48, ‘1’ as 49, etc.

Note the difference between a numeric representation (that can be used in arithmetic calculations) vs. a number represented as a string of digit characters (not used in calculations).



Here, the binary representation of 124 is the binary number for that value. The string representation '124' consists of a longer sequence of bits, eight bits (one byte) for each digit character.

Converting Between a Character and Its Encoding

Python has a means of converting between a character and its encoding.

The ord function gives the UTF-8 (ASCII) encoding of a given character. For example,

`ord('A')` is 65

The chr function gives the character for a given encoding value, thus

`chr(65)` is 'A'

While in general there is no need to know the specific encoding of a given character, there are times when such knowledge can be useful.

Control Characters

Control characters are special characters that are not displayed, but rather *control* the display of output, among other things. Control characters do not have a corresponding keyboard character, and thus are represented by a combination of characters called an ***escape sequence*** .

Escape sequences begin with an ***escape character*** that causes the characters following it to “escape” their normal meaning. **The backslash (\\) serves as the escape character in Python**. For example, **'\n'**, represents the ***newline control character,*** that begins a new screen line,

```
print('Hello\nJennifer Smith')
```

which is displayed as follows:

```
Hello
Jennifer Smith
```

Let’s Try It

What is displayed by each of the following?

```
>>> print('Hello World')
???

>>> print('Hello World\n')
???

>>> print('Hello World\n\n')
???

>>> print('\nHello World')
???
```

```
>>> print('Hello\nWorld')
???

>>> print('Hello\n\nWorld')
???

>>> print(1, '\n', 2, '\n', 3)
???

>>> print('\n', 1, '\n', 2, '\n', 3)
???
```

PES UNIVERSITY

String Formatting

Built-in function format can be used for controlling how strings are displayed, in addition to controlling the display of numerical values,

```
format(value, format_specifier)
```

where ***value*** is the value to be displayed, and ***format_specifier*** can contain a combination of formatting options.

For example, the following produces the string 'Hello' **left-justified in a field width of 20 characters**,

```
format('Hello', '< 20') → 'Hello               '
```

To **right-justify the string**, the following would be used,

```
format('Hello', '> 20') → '               Hello'
```

Formatted strings are left-justified by default. Therefore, the following produce the same result,

```
format('Hello', '< 20') → 'Hello               '
format('Hello', '20') →   'Hello               '
```

PES UNIVERSITY

To **center a string** the **'^'** character is used:

```
format('Hello', '^20')
```

Another use of the format function is to **create a string of blank characters**, which is sometimes useful,

```
format(' ', '15') → '               '
```

Finally **blanks, by default, are the fill character** for formatted strings. However, **a specific fill character can be specified** as shown below,

```
>>> print('Hello', format('.', '.< 30'), 'Have a Nice Day!')
Hello .............................. Have a Nice Day!
```

Here, a single period is the character printed within a field width of 30, therefore ultimately printing out 30 periods.

Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters. There are two ways in Python to deal with such situations:

- **implicit line joining**
- **explicit line joining**

Implicit Line Joining

There are certain delimiting characters that allow a *logical program line* to span more than one *physical line*. This includes matching parentheses, square brackets, curly braces, and triple quotes.

For example, the following two program lines are treated as one logical line:

```
print('Name:',student_name, 'Address:', student_address,
      'Number of Credits:', total_credits, 'GPA:', current_gpa)
```

Matching quotes (except for triple quotes) must be on the same physical line. For example, the following will generate an error:

```
print('This program will calculate a restaurant tab for a couple
      with a gift certificate, and a restaurant tax of 3%')
```

open quote

close quote

Explicit Line Joining

In addition to implicit line joining, program lines may be explicitly joined by use of the backslash (\\) character. Program lines that end with a backslash that are not part of a literal string (that is, within quotes) continue on the following line.

```
numsecs_1900_dob  =  ((year_birth 2 1900) * avg_numsecs_year)  +  \
                     ((month_birth 2 1) * avg_numsecs_month) +  \
                      (day_birth * numsecs_day)
```

- In the display, we observe a space between the output fields.
- By default, output field separator is a space.
- It can be changed by specifying sep = <val> in print.
-
- e) After each print, we get a newline. This is called the output record separator.
- This can be changed by specifying end = <val> in print.

```
# file : 1_output.py
# output
#               output field separator : appears between fields in the output
#                               default :  space
#                               use sep to change this
#               output record seperator : appears at the end of each print
#                               default : newline
#                               use end to change this
""""
print("one", "two", "three")
print("four", "five")

print("one", "two", "three", sep = "-----", end = "\n************\n")
print("four", "five", sep = "^^^^^^^^")
""""
```

```
sep = "stupid"
print("rama", "krishna")
print("apple", "banana", sep = "fool") # does not create a variable
called sep
print(sep)  # stupid and not fool
```

sep = " stupid "

# value of variable sep is substituted and nothing special

print("rama", "krishna", "parama ", "hamsa ", " not ", sep) # sep refers to the variable in the program

print("apple", "banana", "carrot", sep = sep)

# field separator takes the value of the variable sep

Let’s Apply It

Hello World Unicode Encoding

It is a long tradition in computer science to demonstrate a program that simply displays “Hello World!” as an example of the simplest program possible in a particular programming language. In Python, the complete Hello World program is comprised of one program line:

```
print('Hello World!')
```

We take a twist on this tradition and give a Python program that displays the Unicode encoding for each of the characters in the string “Hello World!” instead. This program utilizes the following programming features:

- **string literals**
- **print**
- **ord function**

```
1  # This program displays the Unicode encoding for 'Hello World!'
2
3  # Program greeting
4  print("The Unicode encoding for 'Hello World!' is:")
5
6  # output results
7  print(ord('H'), ord('e'), ord('l'), ord('l'), ord('o'), ord(' '),
8        ord('W'), ord('o'), ord('r'), ord('l'), ord('d'), ord('!'))
```

Program Execution ...

```
The Unicode encoding for 'Hello World!' is:
72 101 108 108 111 32 87 111 114 108 100 33
```

The statements on **lines 1, 3,** and **6** are **comment statements**. The **print function** on **line 4** displays the message ‘Hello World!’. **Double quotes** are used to delimit the corresponding string, since the single quotes within it are to be taken literally. The use of print on **line 7** **prints out the Unicode encoding**, one-by-one, for each of the characters in the “Hello World!” string. Note from the program execution that there is a Unicode encoding for the blank character (32), as well as the exclamation mark (33).

Variables and Identifiers

So far, we have only looked at literal values in programs. However, **the true usefulness of a computer program is the ability to operate on different values each time the program is executed**. This is provided by the notion of a ***variable***. We look at variables and identifiers next.

What Is an Identifier?

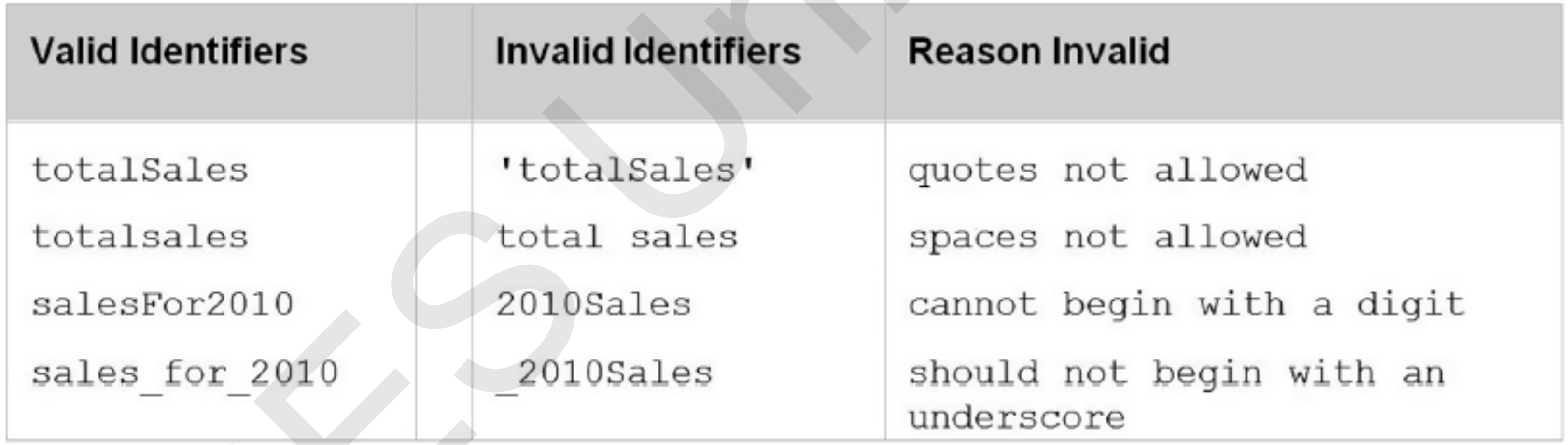
An **identifier** is a sequence of one or more characters used to provide a name for a given program element. Variable names `line`, `num_credits`, and `gpa` are each identifiers.

Python is **case sensitive** , thus, `Line` is different from `line`. Identifiers may contain letters and digits, but cannot begin with a digit.

The **underscore character**, _, is also allowed to aid in the readability of long identifier names. **It should not be used as the first character**, however, as identifiers beginning with an underscore have special meaning in Python.

Spaces are not allowed as part of an identifier. This is a common error since some operating systems allow spaces within file names. In programming languages, however, **spaces are used to delimit (separate) distinct syntactic entities**. Thus, any identifier containing a space character would be considered two separate identifiers.

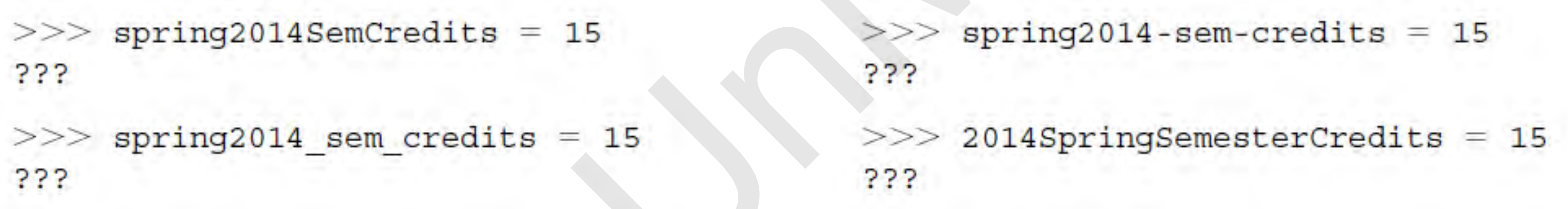
Examples of valid and invalid identifiers in Python



Valid Identifiers		Invalid Identifiers	Reason Invalid
totalSales		'totalSales'	quotes not allowed
totalsales		total sales	spaces not allowed
salesFor2010		2010Sales	cannot begin with a digit
sales_for_2010		_2010Sales	should not begin with an underscore

Let’s Try It

What is displayed by each of the following?



```
>>> spring2014SemCredits = 15
???
```

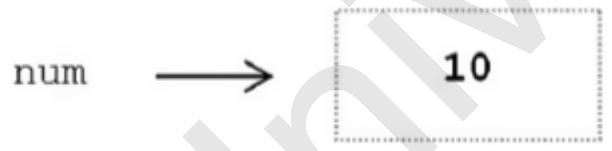
```
>>> spring2014_sem_credits = 15
???
```

```
>>> spring2014-sem-credits = 15
???
```

```
>>> 2014SpringSemesterCredits = 15
???
```

What Is a Variable?

A variable is a name (identifier) that is associated with a value,



A variable can be assigned different values during a program's execution—hence, the name "variable."

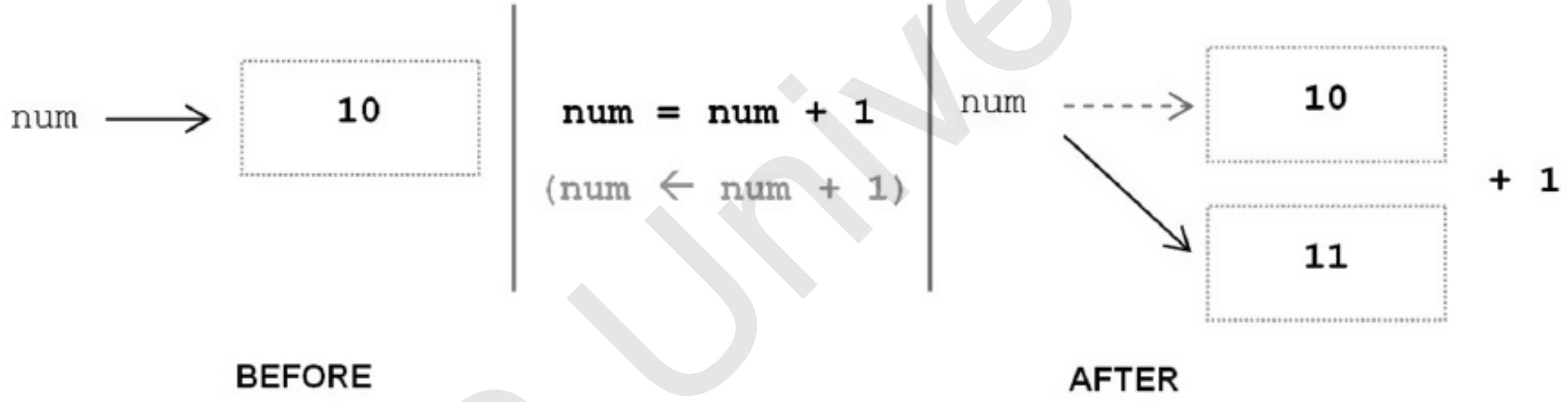
Wherever a variable appears in a program (except on the left-hand side of an assignment statement), **it is the value associated with the variable that is used**, and not the variable's name,

```
num + 1 → 10 + 1 → 11
```

Variables are assigned values by use of the **assignment operator , = ,**

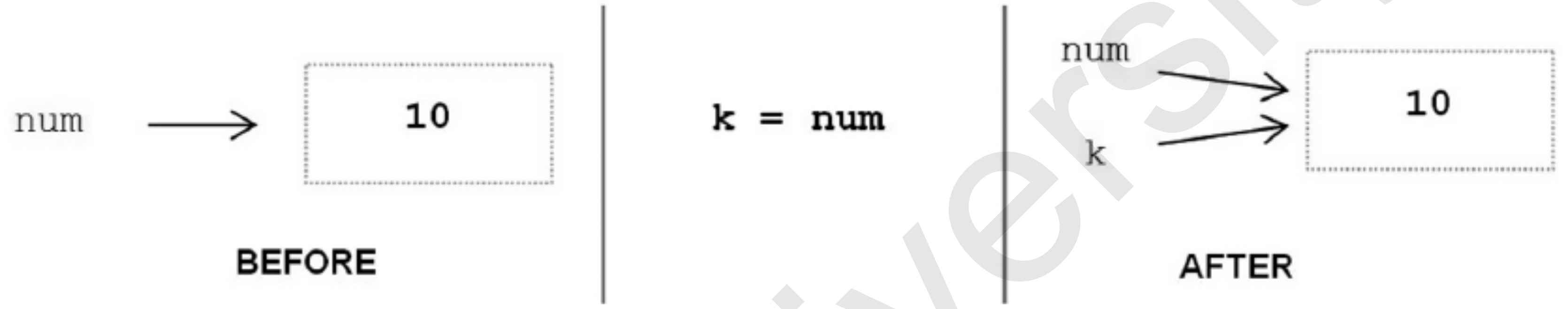
```
num = 10        num = num + 1
```

Assignment statements often look wrong to novice programmers. Mathematically, **num = num + 1** does not make sense. In computing, however, it is used to **increment** the value of a given variable by one. It is more appropriate, therefore, to think of the **=** symbol as an arrow symbol

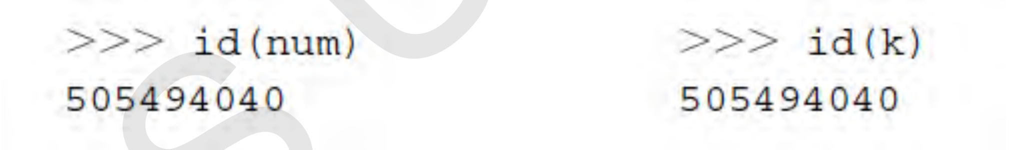


When thought of this way, it makes clear that **the right side of an assignment is evaluated first, then the result is assigned to the variable on the left**. An arrow symbol is not used simply because there is no such character on a standard computer keyboard.

Variables may also be assigned to the value of another variable,



Variables **num** and **k** are both associated with the same literal value 10 in memory. One way to see this is by use of **built-in function** id,

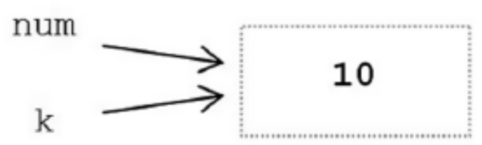


```
>>> id(num)
505494040
```

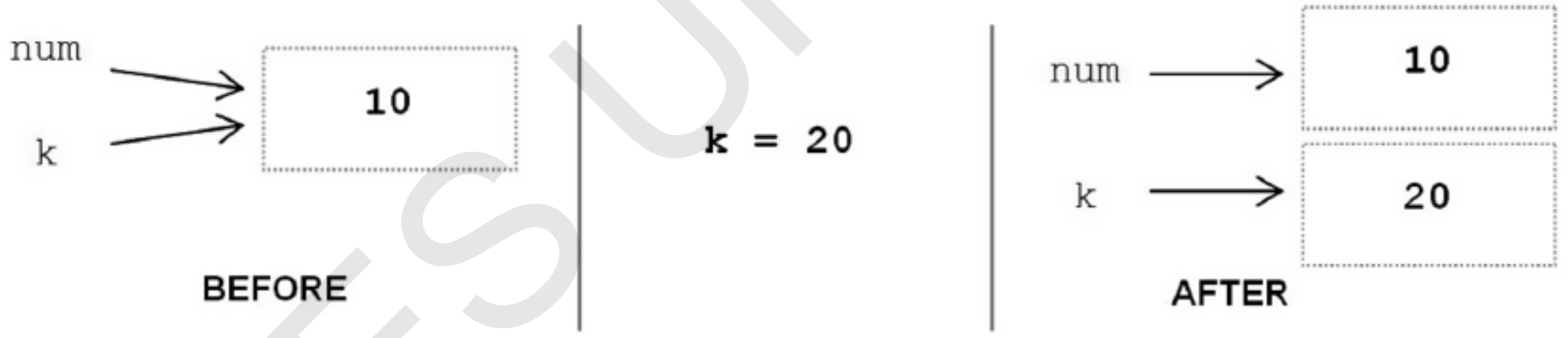
```
>>> id(k)
505494040
```

The id function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems.

If the value of **num** changed, would variable **k** change along with it?



This cannot happen in this case because the variables refer to integer values, and integer values are *immutable*. An **immutable value** is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned,



If no other variable references the memory location of the original value, the memory location is ***deallocated*** *(that is, it is made available for reuse).*

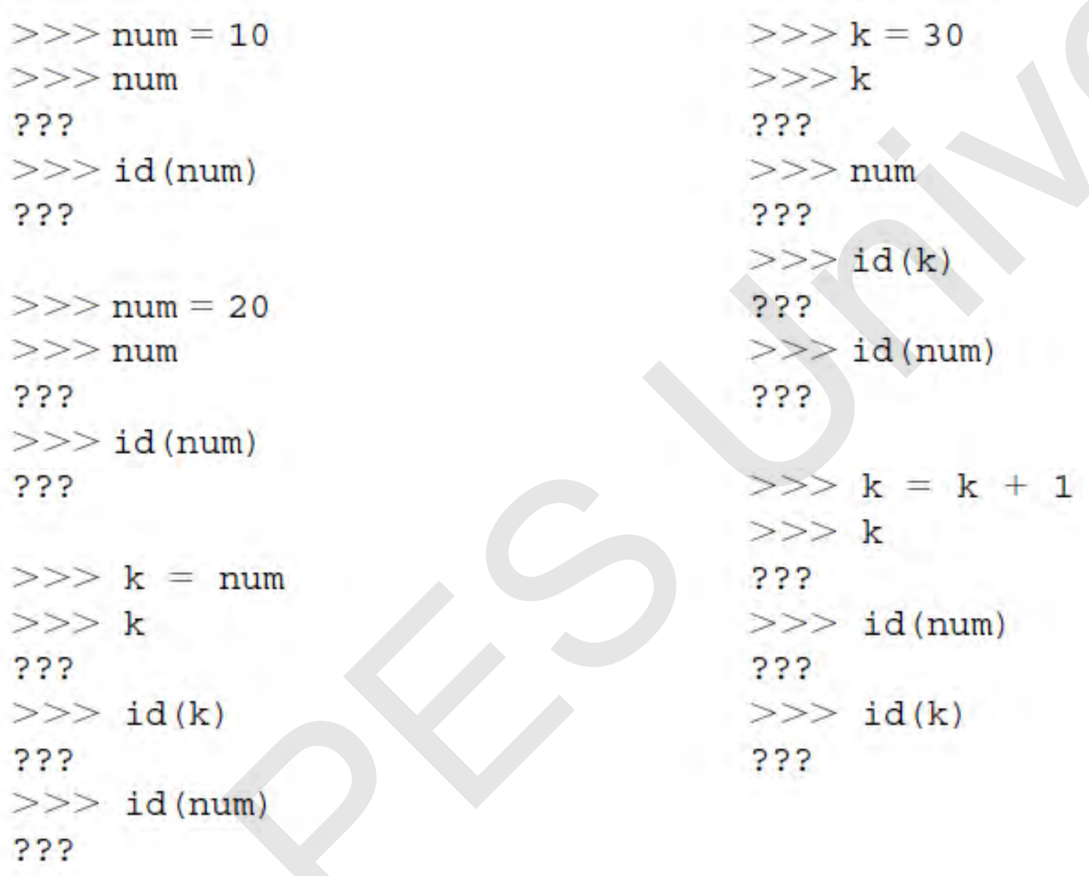
Finally, in Python **the same variable can be associated with values of different type** during program execution,

```
var = 12               integer
var = 12.45            float
var = 'Hello'          string
```

(The ability of a variable to be assigned values of different type is referred to as ***dynamic typing***, introduced later in the discussions of data types.)

Let’s Try It

What do each of the following evaluate to?



```
>>> num = 10
>>> num
???
>>> id(num)
???

>>> num = 20
>>> num
???
>>> id(num)
???

>>> k = num
>>> k
???
>>> id(k)
???
>>> id(num)
???
```

```
>>> k = 30
>>> k
???
>>> num
???
>>> id(k)
???
>>> id(num)
???

>>> k = k + 1
>>> k
???
>>> id(num)
???
>>> id(k)
???
```

PES UNIVERSITY

Variable Assignment and Keyboard Input

The value that is assigned to a given variable does not have to be specified in the program. The value can come from the user by use of the **input function** introduced in Chapter 1,

```
>>> name = input('What is your first name?')
What is your first name? John
```

In this case, the variable name is assigned the string `'John'`. If the user hit return without entering any value, name would be assigned to the **empty string** (`''`).

The input function returns a string type. For input of numeric values, the response must be converted to the appropriate type. Python provides built-in **type conversion functions int()** and **float ()** for this purpose,

```
line = input('How many credits do you have?')
num_credits = int(line)
line = input('What is your grade point average?')
gpa = float(line)
```

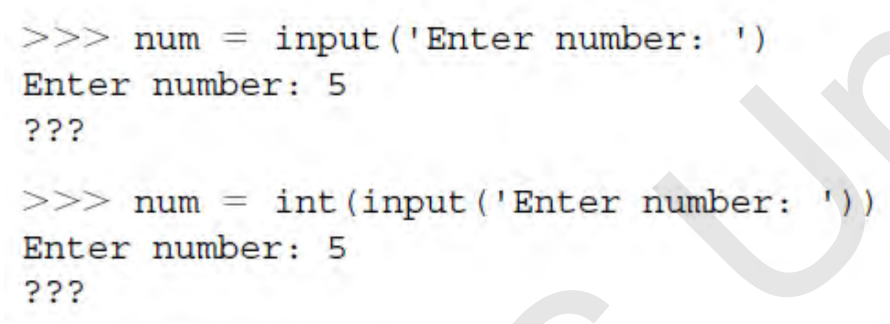
The entered number of credits, say `'24'`, is converted to the equivalent integer value, `24`, before being assigned to variable `num_credits`. The input value of the gpa, say `'3.2'`, is converted to the equivalent floating-point value, `3.2`.

Note that the program lines above could be combined as follows,

```
num_credits = int(input('How many credits do you have? '))
gpa = float(input('What is your grade point average? '))
```

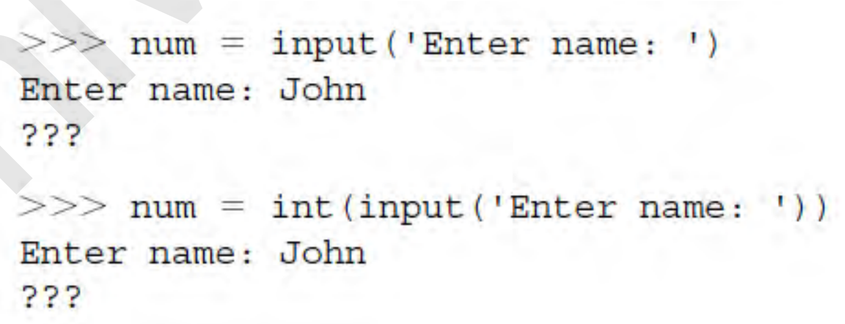
Let’s Try It

What is displayed by each of the following?



```
>>> num = input('Enter number: ')
Enter number: 5
???
```

```
>>> num = int(input('Enter number: '))
Enter number: 5
???
```



```
>>> num = input('Enter name: ')
Enter name: John
???
```

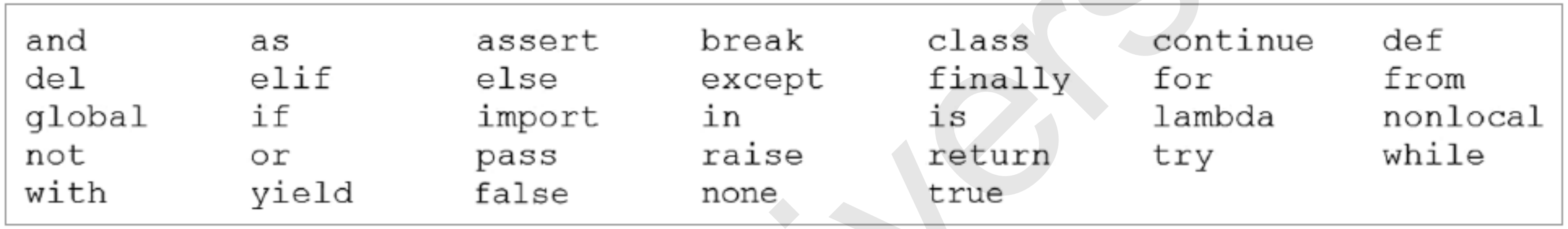
```
>>> num = int(input('Enter name: '))
Enter name: John
???
```

Keywords and Other Predefined Identifiers in Python

A **keyword** is an identifier that has predefined meaning in a programming language. Therefore, **keywords cannot be used as "regular" identifiers**. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword **and** below,

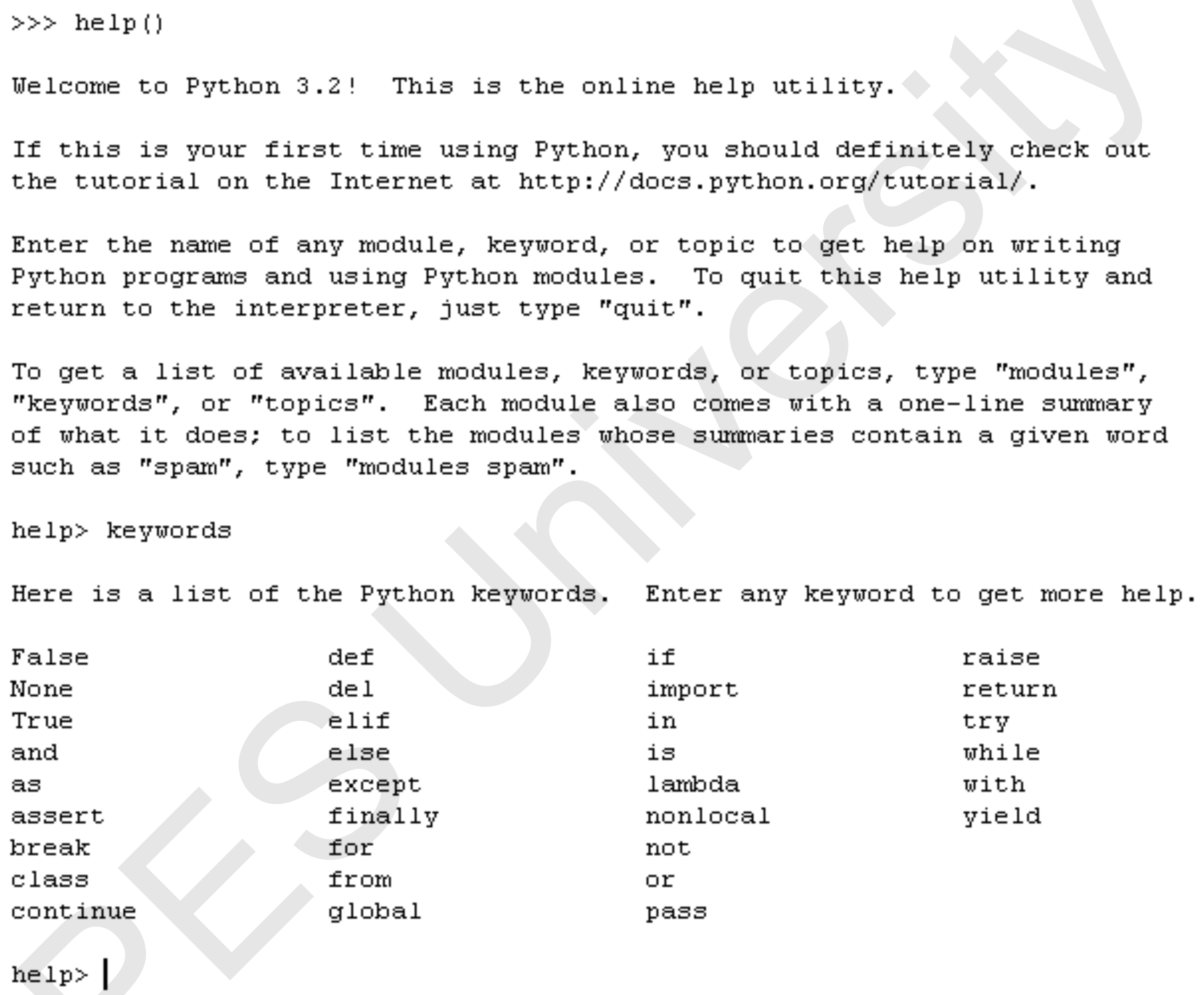
```
>>> and = 10

SyntaxError: invalid syntax
```

The keywords in Python are listed below.



and	as	assert	break	class	continue	def
del	elif	else	except	finally	for	from
global	if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try	while
with	yield	false	none	true		

To display the keywords in Python, type **`help()`** in the Python shell, then type **`keywords`** (type 'q' to quit).



```
>>> help()

Welcome to Python 3.2!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass

help> |
```

Let’s Try It

What is displayed by each of the following?

```
>>> yield = 1000
???

>>> Yield == 1000
???
```

```
>>> print('Hello')
???

>>> print = 10
>>> print('Hello')
???
```

Variables

One of the most fundamental concepts in programming is that of a **variable**.

A variable is "**a name that is assigned to a value**," as shown below,

```
n = 5
```
variable `n` is assigned the value 5

Thus, whenever variable `n` appears in a calculation, it is the current value of n is that is used, as in the following,

```
n + 20
```
(5 + 20)

If variable `n` is assigned a new value, then the same expression will produce a different result,

```
n = 10
n + 20
```
(10 + 20)

Some Basic Arithmetic Operators

The **common arithmetic operators** in Python are,

+ (**addition**)	* (**multiplication**)	** (**exponentiation**)
– (**subtraction**)	/ (**division**)	

Addition, subtraction, and division use standard mathematical notation,

```
10 + 20          25 - 15          20 / 10
```

(Also, // for truncated division, discussed later)

For multiplication and exponentiation, the asterisk (*) is used,

`5 * 10` (5 times 10)   `2 ** 4` (2 to the 4th power)

Multiplication is never denoted by the use of parentheses,

`10 * (20 + 5)` CORRECT   `10(20 + 5)` INCORRECT

Note that parentheses may be used to denote subexpressions.

Type

#type()-built in function

#type(object)

#it returns type of the given object.

#type of a variable depends on the value assigned to it

a = 10

print(type(a))

# int a = 10.0

print(type(a)) # float

```
a = "python“
print(type(a)) # str
a = True
print(type(a))  # values of bool type : False True
a=False
print(type(a))
a = 2 + 3j
print(type(a)) # complex
```

The below ones are called as structured or reference types: having more than one value.

```
a = (10, 20, 30, 40)
print(type(a)) # tuple

a = [10, 20, 30, 40]
print(type(a)) # list

a = {10, 20, 30, 40}
print(type(a)) # set

a = {1:20,2:30}
print(type(a)) # dict
```