ICS 104 - Introduction to Programming in Python and C

Programming with numbers and Strings

Reading Assignment

- Chapter 2 Sections 1, 2, 4 and 5.

Chapter Learning Outcomes

At the end of this chapter, you will be able to

- define and use variables and constants
- write arithmetic expressions and assignment statements
- understand the properties and limitations of integers and floating-point numbers
- appreciate the importance of comments and good code layout
- write arithmetic expressions and assignment statements
- create programs that read and process inputs, and display the results
- learn how to use Python strings

Variables

Why do we need variables?

- To carry out computation, we need to store values in order to use them later on.
- These values are stored in variables.
- Let us try to comprehend the use of variables by solving the following problem:

Soft Drinks: Which is more Economic?

- Soft drinks are sold in cans and bottles.
- A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle.
- Find the volume (in liters) of a six-pack of soda cans and the total volume of a six-pack and a two-liter bottle.
    - Note that 12 fluid ounces equal approximately 0.355 liters.

Defining Variables

- A variable is a storage location in a computer program.
- Each variable has a name and holds a value.

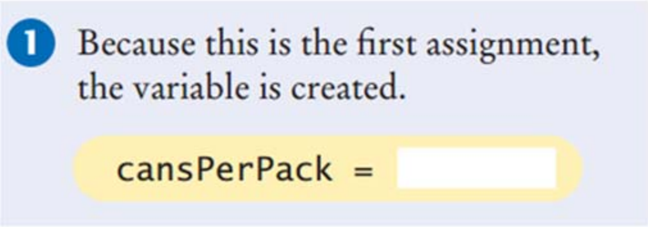
- Just as a parking space has an identifier **J053** and contents **car**

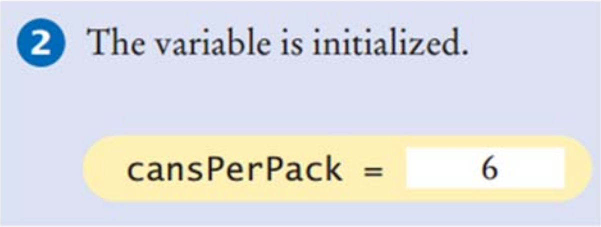
Assignment Statements

- An **assignment** statement is used to place a value into a variable

```
In [ ]:
 cansPerPack = 6
```

- How does the assignment statment work?
  - The right hand side of the **=** sign is first evaluated (to the value 6).
  - The value is assigned to the variable on the left hand side of the **=** sign (to the variable **cansPerPack**).


1 Because this is the first assignment, the variable is created.


2 The variable is initialized.

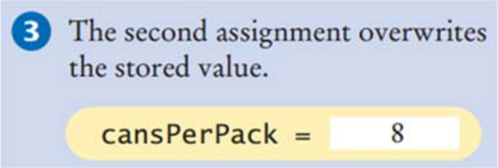
- Once a variable is defined, it can be used in other statements

```
In [ ]:
 print(cansPerPack)
```

- If an existing variable is assigned a **new** value, that value replaces the previous contents of the variable.

```
In [ ]:
 cansPerPack = 8
```

```
In [ ]:
 print(cansPerPack)
```


3 The second assignment overwrites the stored value.

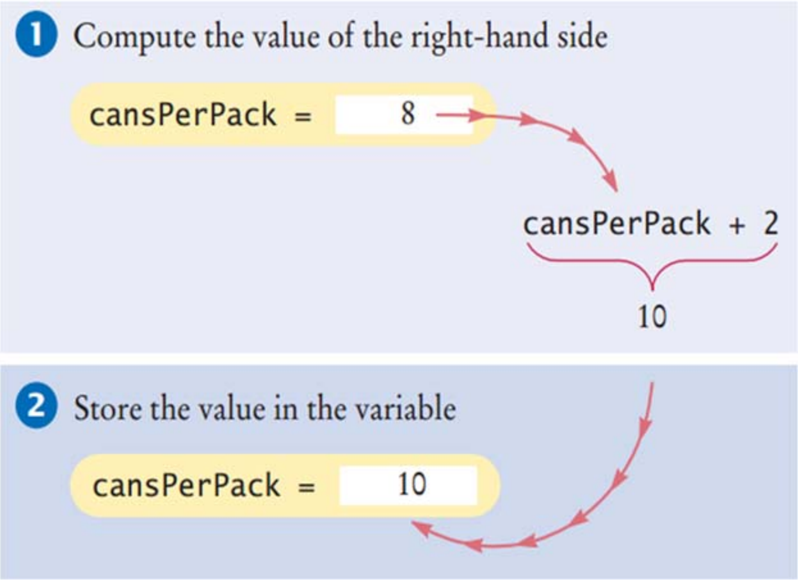
Assignment is not Equality in Algebra

- Is the statement
  - `cansPerPack = cansPerPack + 2`

  correct in Algebra?
- How about in Python?

```
In [ ]:
 cansPerPack = 8
 cansPerPack = cansPerPack + 2
 print(cansPerPack)
```

- So, how does the assignment cansPerPack = cansPerPack + 2 execute in python?
- First, the right hand side is executed
  - This is done by fetching the current value of the variable **cansPerPack**
  - Then, carrying out the addition
- Second, the value of the addition is stored in the variable **cansPerPack**


1 Compute the value of the right-hand side
cansPerPack = 8
cansPerPack + 2
10
2 Store the value in the variable
cansPerPack = 10

Number Types

Values and Types

- **2**, **"Hello World"** and **8.4** are **values**
- Each value belongs to a **data type**

    - **2** is an **integer** **int**
    - **"Hello World"** is a **string** **str**
    - **8.4** is a **float** **float**
    - **2** and **8.4** are called **number literals**.

Why Data Types?

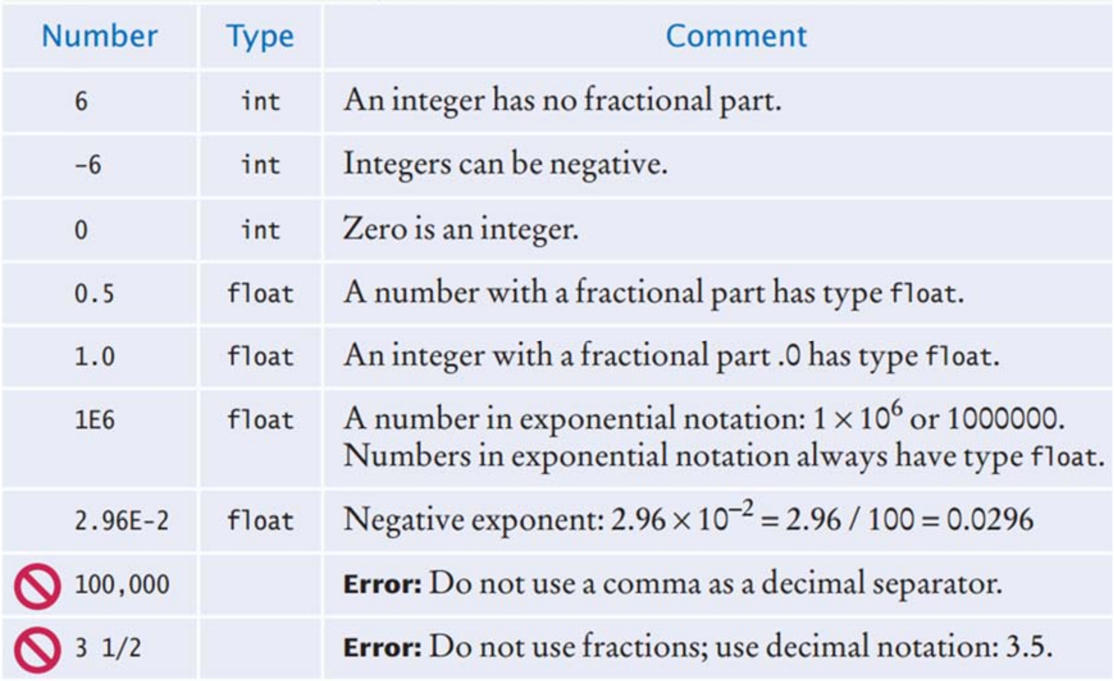
- A **data type** of a value determines

    - how the data type is represented in the computer, and
    - what operations can be performed on that data.

Two Categories of Data Types in Python

- Primitive data type
    - A data type provided by the language itself (e.g. **int**)
- User-defined data type
    - A data type defined by the programmer (covered in Chapter 9: **Objects and Classes**)

Number literals in Python



Number	Type	Comment
6	int	An integer has no fractional part.
-6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	float	A number with a fractional part has type float.
1.0	float	An integer with a fractional part .0 has type float.
1E6	float	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type float.
2.96E-2	float	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
🚫 100,000		**Error:** Do not use a comma as a decimal separator.
🚫 3 1/2		**Error:** Do not use fractions; use decimal notation: 3.5.

- The value determines the type of the variable.
- For example, the following piece of code is correct, but not recommended

In [ ]:

```
taxRate = 5
print(taxRate)
taxRate = 5.5
print(taxRate)
taxRate = "five point five"
print(taxRate)
```

- This is not a good idea, as it may lead to an error if you use the wrong operation on the variable

In [ ]:

```
taxRate = taxRate + 10
```

- Once a variable is initialized with a value of a type, keep storing values of the same type.

Rules for Variable Names

- Names must start with a letter or the underscore (_) character.
- The remaining characters (if any) must be letters, digits or underscores.
  - Symbols such as ? or % cannot be used in a variable name.
  - Spaces cannot exist within a variable name.
- Names are case sensitive.
- Reserved words by python cannot be used as variable names. (e.g., **if** and **class**)

- Which of the following names are proper variable names? `canVolume1 , x , CanVolume , 6pack , can volume , class , ltr/fl.oz`

- `canVolume1` is proper

- `x` is proper

- `CanVolume` is proper

- `6pack` is not proper

- `can volume` is not proper

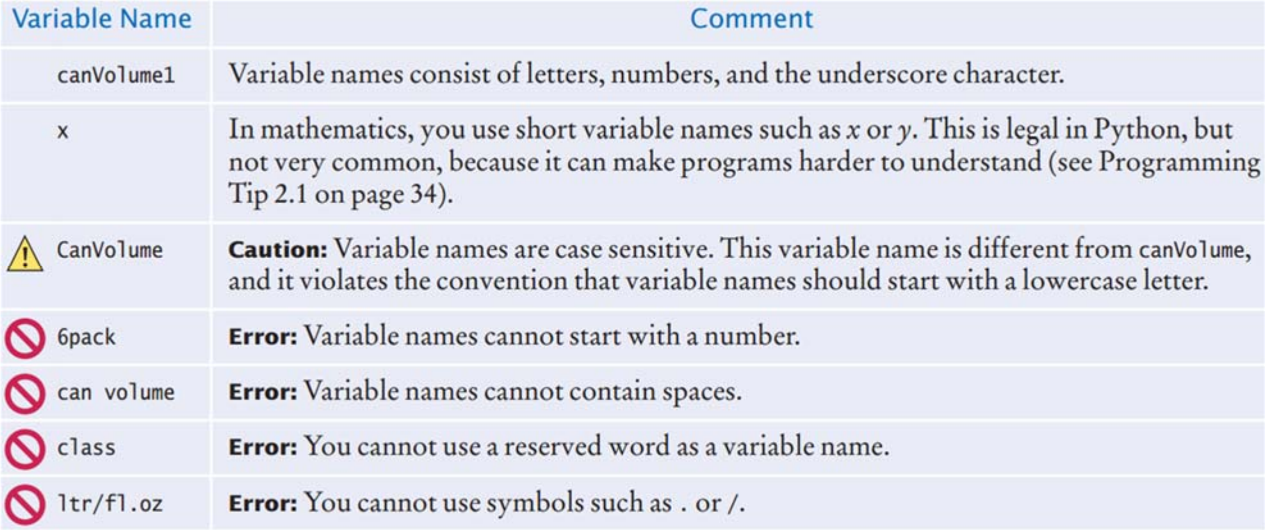
- `class` is not proper

- `ltr/fl.oz` is not proper

Recommended Variable Name Conventions

- These are not strict rules for variable names, but are **rules of good taste** that you should respect when writing code.
  - Use a descriptive name, such as **`cansPerPack`**, than a terse name, such as **`cpp`**
    - If the variable name consists of more than one word, start the word with a capital letter, as shown above.
  - A variable starts with a small letter
  - A constant consists of all capital letters, where words are separated by the underscore _ character, such as **`CAN_VOLUME`**
  - A user defined data type starts with a capital letter (as we will see later), such as **`GraphicsWindow`**.

Therefore,



	Variable Name	Comment
	`canVolume1`	Variable names consist of letters, numbers, and the underscore character.
	`x`	In mathematics, you use short variable names such as *x* or *y*. This is legal in Python, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 34).
⚠	`CanVolume`	**Caution:** Variable names are case sensitive. This variable name is different from `canVolume`, and it violates the convention that variable names should start with a lowercase letter.
🚫	`6pack`	**Error:** Variable names cannot start with a number.
🚫	`can volume`	**Error:** Variable names cannot contain spaces.
🚫	`class`	**Error:** You cannot use a reserved word as a variable name.
🚫	`ltr/fl.oz`	**Error:** You cannot use symbols such as `.` or `/`.

Constants

- A constant variable, or simply a **constant**, is a variable whose value should not be changed after it has been assigned an initial value.
- Some languages provide an explicit mechanism of declaring constants.
  - Hence, any attempt to change it after it has been assigned generates a syntax error.
- Python leaves it to the programmer to make sure that constants are not changed.
  - Hence, the use of all capital letters for naming constants tells you and other programmers that you should not change the value of this **variable** once it is assigned.

- Constants can make your code much more understandable.

- For example, compare the following two statements:
  - `totalVolume = bottles * 2`
  - `totalVolume = bottles * BOTTLE_VOLUME`
- Note that in the case where the bottle volume is changed from 2 to 2.5, then
  - in the first case, you need to change every line of code that has volume 2 to 2.5.
  - in the second case, all you need to do is change the value of the constant **BOTTLE_VOLUME** to 2.5 in one line ONLY. Every other occurrence of **BOTTLE_VOLUME** in the code will `automatically` have the new volume value.

Comments

- As your programs get more complex, you should add **comments**, *explanations for human readers of your code*.

In [ ]:

```
CAN_VOLUME = 0.355   # Liters in a 12-ounce can
```

- This comment explains the significance of the value 0.355 to a human reader.

- Python's interpreter does not execute comments at all.
    - It ignores everything from a **#** delimiter to the end of the line.

Why Write Comments?

- Helps programmers who read your code understand your intent.

- Helps you when you review your code (after some time).

How to Write Comments?

- Provide a comment at the top of your source file that explains the purpose of the program.

- The textbook follows the following style:

In [ ]:

```
##
# This program computes the volume (in liters) of a six-pack of soda cans.
#
```

Time to Solve the Problem at the Beginning of this Chapter

Soft Drinks: Which is more Economic?

- Soft drinks are sold in cans and bottles.
- A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle.
- Which one should you buy?

Solution Steps

- Compute the **totalVolume** you get when you buy a six-pack
    - Define **CAN_VOLUME** and the number of **cansPerPack**
    - `totalVolume = cansPerPack * CAN_VOLUME`
    - print the **totalVolume**

- Now you can compare the **totalVolume** to the value **2.0** and determine which one to buy

In [ ]:

```
##
#  This program computes the volume (in liters) of a six-pack of soda
#  cans and the total volume of a six-pack and a two-liter bottle.
#

# Liters in a 12-ounce can and a two-liter bottle.
CAN_VOLUME = 0.355
BOTTLE_VOLUME = 2

# Number of cans per pack.
cansPerPack = 6

# Calculate total volume in the cans.
totalVolume = cansPerPack * CAN_VOLUME
print("A six-pack of 12-ounce cans contains", totalVolume, "liters.")

# Calculate total volume in the cans and a two-liter bottle.
totalVolume = totalVolume + BOTTLE_VOLUME
print("A six-pack and a two-liter bottle contain", totalVolume, "liters.")
```

Final Tips on Variables

- Do not use undefined variables
    - `canVolume = 12 * literPerOunce # Error`
    - `literPerOunce = 0.0296`

- Choose descriptive variable names
    - **canVolume** is better than **cv**

- Do not use *magic* numbers
    - `totalVolume = cansPerPack * 0.355`

2.2 Arithmetic

Basic Arithmetic Operations

- Python supports addition `+`, subtraction `-`, multiplication `*` and division `/`
- **`+ - * /`** are called **operators**
- The combination of variables, literals, operators, and parentheses is called an arithmetic **expression**
- For example, the mathematical formula a+b2a+b2 is written in python as `(a + b) / 2`
  - Note that the parentheses are used to determine in which order the parts of the expression are computed.
  - For example, which mathematical formula is `a + b / 2`?
- Python uses the exponential operator **** to denote the power operation.
  - For example, a2a2 is `a ** 2`

Precedence of Arithmetic Operators

- Python uses the precedence rules for algebraic notation

Precedence	Operator(s)	Description
1	()()	Parentheses
2	****	Power
3	*,/*,/	Multiplication and Division
4	+,−+,−	Addition and Subtraction

Order of Evaluation of Arithmetic Operators

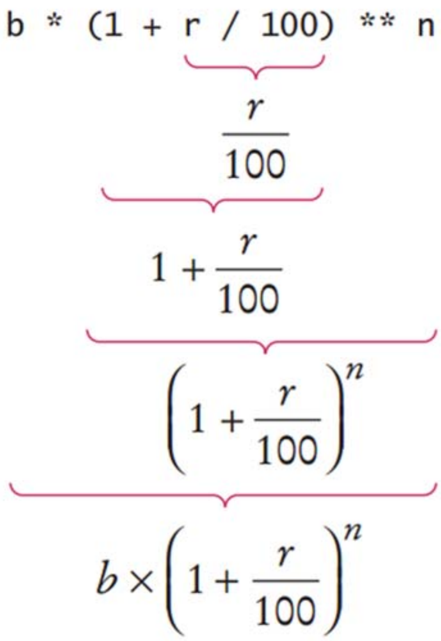
- Addition, subtraction, multiplication and division are left associative, i.e. they are evaluated from left to right.
  - For example, `10 + 2 + 3` is evaluated as (10+2)+3=15(10+2)+3=15
- The power operation is right associative, i.e. it is evaluated from right to left.
  - For example, `10 ** 2 ** 3` is evaluated as 10231023 which is the same as 108=100000000108=100000000

Example

- The mathematical expression b×(1+r100)nb×(1+r100)n becomes

```
b * (1 + r / 100) ** n
```

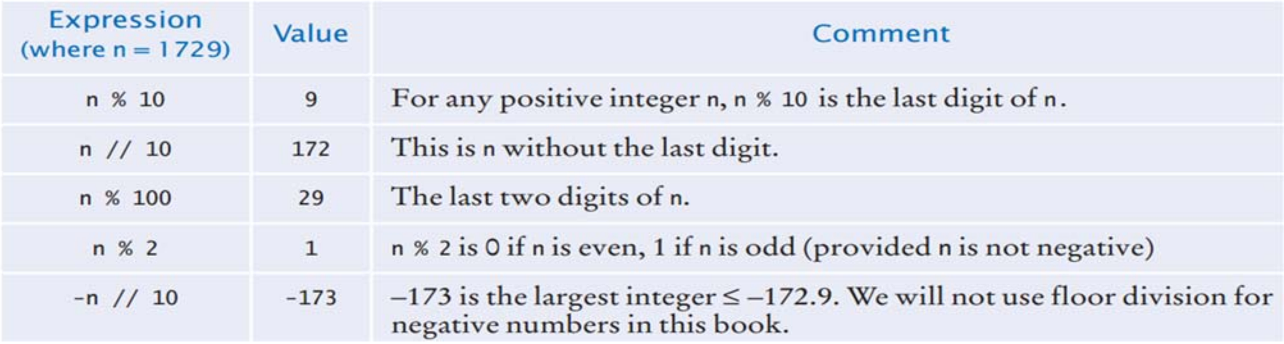
- The expression is analyzed as follows


b * (1 + r / 100) ** n
$\frac{r}{100}$
$1+\frac{r}{100}$
$\left(1+\frac{r}{100}\right)^n$
$b\times\left(1+\frac{r}{100}\right)^n$

Floor Division and Remainder

- Division of two integers results in a floating-point value
  - `7 / 4` yields `1.75`
- The floor division operator `//` when applied on positive integers computes the quotient and discards the fractional part.
  - `7 // 4` yields 1
- The **modulus** operator `%` can be used to get the remainder of the floor division.
  - `7 % 4` yields 3, the remainder of the floor division of 7 by 4.
  - Some also call it **modulo** or **mod**

Floor Division and Remainder



Expression (where n = 1729)	Value	Comment
`n % 10`	9	For any positive integer n, n % 10 is the last digit of n.
`n // 10`	172	This is n without the last digit.
`n % 100`	29	The last two digits of n.
`n % 2`	1	n % 2 is 0 if n is even, 1 if n is odd (provided n is not negative)
`-n // 10`	-173	−173 is the largest integer ≤ −172.9. We will not use floor division for negative numbers in this book.

Calling Functions

- We have been using the print function to display information, but there are many other functions available in Python.
- Most functions **return** a value.
  - i.e., when the function completes its task, it passes a value back to the point where the function was called.
  - For example, the call `abs(-123)` returns the value 123.
- The value returned by a function can be stored in a variable.
  - `distance = abs(x)`
  - Note that `x` is called the **argument** of the **abs** function.
- It can also be used anywhere that a value of the same type can be used
  - `print("The distance from the origin is ", abs(x))`

Arguments of a Function

- When calling a function, you must provide the correct number of arguments.
  - `abs(-10, 2)` or `abs()` will generate an error.
  - Hence, the **`abs`** function requires exactly one argument.

```
In [ ]:
 abs(-10)
```

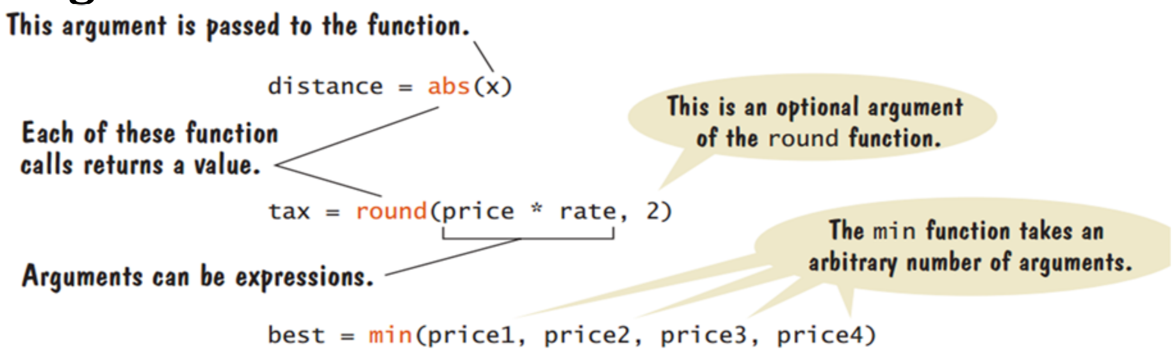
- Some functions have optional arguments that you only provide in certain situations
  - For example, in the **`round`** function
    - o `round(7.625)` returns the nearest integer, i.e. `8`
    - o `round(7.625,2)` returns the nearest floating-point with 2 decimal digits, i.e. `7.63`

```
In [ ]:
 round(7.625)
```

- Some functions take an arbitrary number of arguments
  - For example, the **`max`** and **`min`** functions.
    - o `min(7.25, 10.95, 5.95, 6.05, 8)` returns the minimum of the function's arguments; in this case the number `5.95`

```
In [ ]:
 min(7.25, 10.95, 5.95, 6.05, 8)
```

Calling Functions



This argument is passed to the function.

Each of these function calls returns a value.

This is an optional argument of the round function.

Arguments can be expressions.

The min function takes an arbitrary number of arguments.

```
distance = abs(x)

tax = round(price * rate, 2)

best = min(price1, price2, price3, price4)
```

Libraries

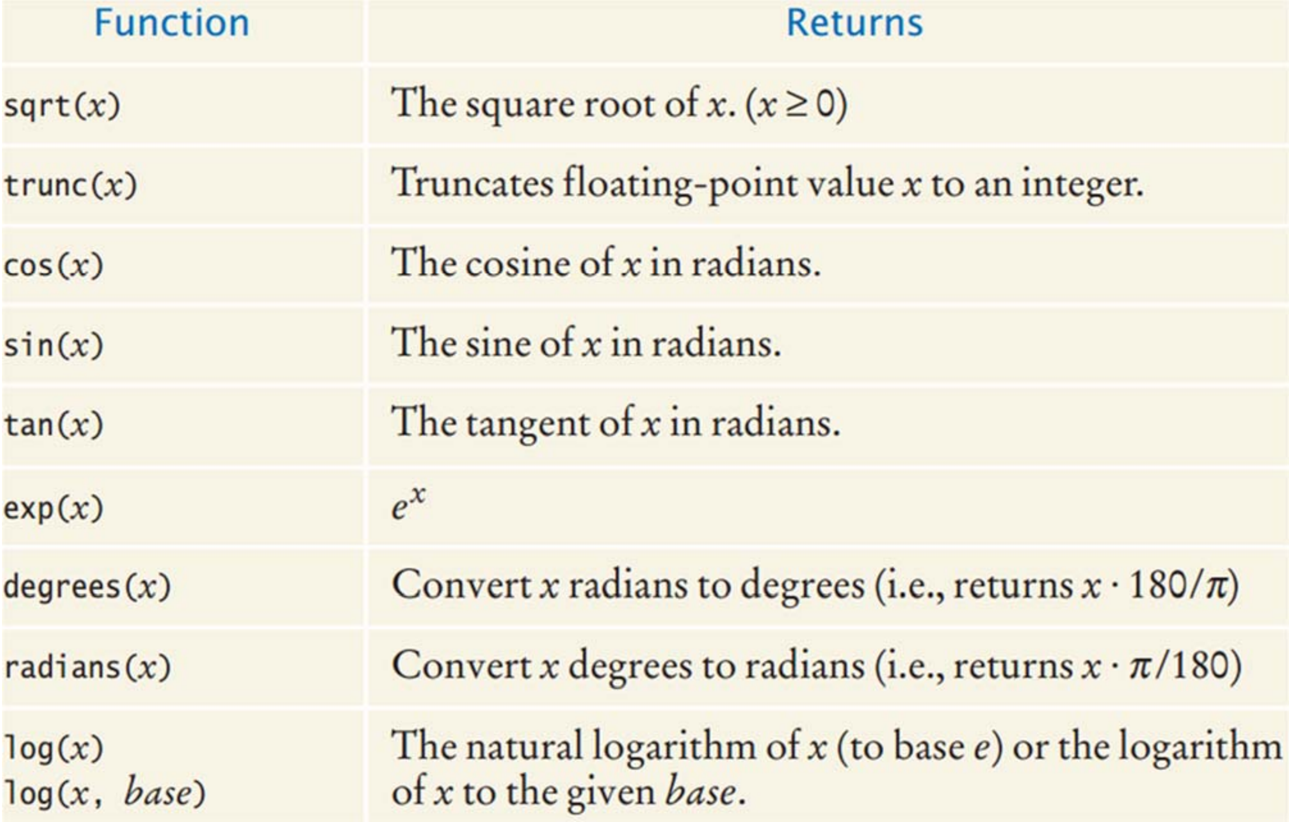
- A **library** is a collection of code that has been written and translated by someone else, ready for you to use in your program.
  - A **standard library** is a library that is considered part of the language and must be included with any Python system.

- Python’s standard library is organized into modules.
  - Related functions and data types are grouped into the same module.

Mathematical Functions

- Python's **math** module includes a number of mathematical functions.
- You must **import** it before you can use any of its functions
    - Note that you can use the `print` function without the use of **import**, since it is one of the **built-in** functions (part of the Python language and can be used directly in your programs).

```
In [ ]:
from math import sqrt
y = sqrt(25)
print("y = ", y)
```

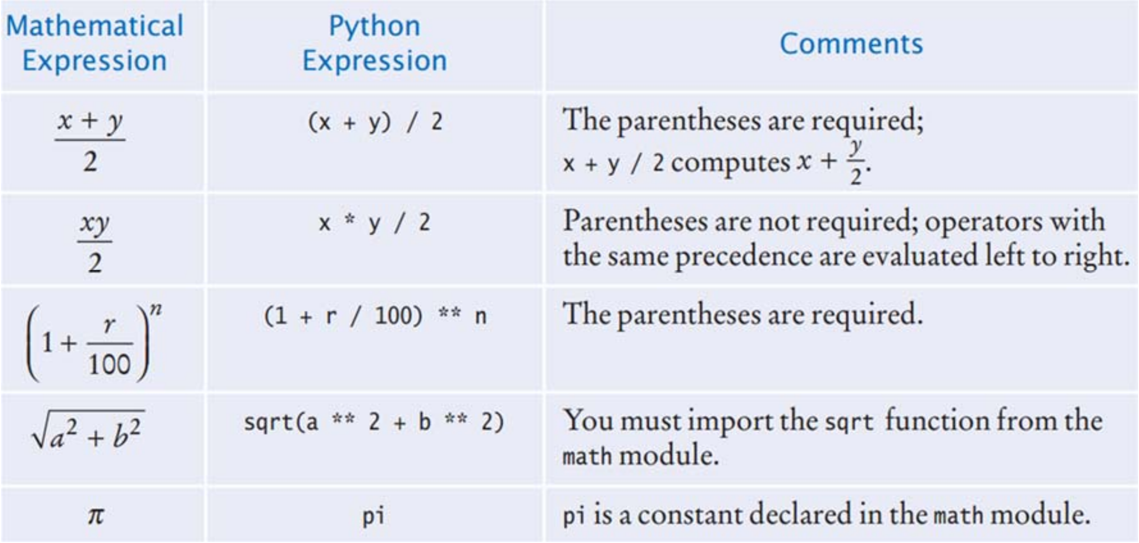


Function	Returns
`sqrt(`x`)`	The square root of x. ($x \geq 0$)
`trunc(`x`)`	Truncates floating-point value x to an integer.
`cos(`x`)`	The cosine of x in radians.
`sin(`x`)`	The sine of x in radians.
`tan(`x`)`	The tangent of x in radians.
`exp(`x`)`	e^x
`degrees(`x`)`	Convert x radians to degrees (i.e., returns $x \cdot 180/\pi$)
`radians(`x`)`	Convert x degrees to radians (i.e., returns $x \cdot \pi/180$)

`log(`x`)`
`log(`x`,` *base*`)`
	The natural logarithm of x (to base e) or the logarithm of x to the given *base*.

- To import more than one function from **math**, use `from math import *`

Arithmetic Expressions Examples



Mathematical Expression	Python Expression	Comments
$\frac{x+y}{2}$	`(x + y) / 2`	The parentheses are required; `x + y / 2` computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	`x * y / 2`	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1+\frac{r}{100}\right)^n$	`(1 + r / 100) ** n`	The parentheses are required.
$\sqrt{a^2+b^2}$	`sqrt(a ** 2 + b ** 2)`	You must import the `sqrt` function from the `math` module.
π	`pi`	`pi` is a constant declared in the `math` module.

Student Activity

- The volume of a sphere is given by

  V=43πr3V=43πr3

If the radius is given by a variable **radius** that contains a floating-point value, write a Python expression for the volume.

In [ ]:

```
# Volume Expression
radius = 2.4
```

2.4 Strings

- A **string** is a sequence of characters
  - **Characters** include letters, numbers/digits, punctuation, spaces, special symbols and so on.
- A **string literal** denotes a particular string (e.g. "Hello")
  - Just as a number literal (e.g. `34`) denotes a particular number.
  - String literals are specified by enclosing a sequence of characters within a matching pair of either single or double quotes.

In [ ]:

```
print("This is a string. ", 'So is this.')
```

- How can I form the strings **`I'm a student`** or **`He said: "You did it!"`**?

In [ ]:

```
print("I'm a student", 'He said: "You did it!"')
```

- The number of characters in a string is called the **length** of the string.
  - For example, "Harry" is of length ______ and "World" is of length _______
  - An **empty** string is a string with no characters. It is of length zero and is written as `""` or `''`

- Python's `len` function returns the length of the argument string.

In [ ]:

```
length = len("World!")
print(length)
```

String Concatenation

- Given two strings such as `Ahmad` and `Saleem`, you can **concatenate** them to one long string.

In [ ]:

```
firstName = "Ahmad"
secondName = "Saleem"
name = firstName + secondName
print (name)
```

- Note that if one of the operands of the `+` operator is a string, then all of them should be strings, otherwise a syntax error will occur.

In [ ]:

```
print("The character with value 7 is the ", chr(1710))
```

String Repetition

- Given a string such as `-`, you can repeat it **n** times, where **n** is an integer using the string repetition operator `*`

In [ ]:

```
dashes = "-" * 50
print(dashes)
```

Converting between Numbers and Strings

- Since you cannot concatenate a string and integer, Python provides the `str` function to convert an integer to a string.

In [ ]:

```
id = 2019873410
id=- 1
email = "s" + str(id) + "@kfupm.edu.sa"
print(email)
```

- Conversely, you can turn a string representing a number into its corresponding numerical value using the `**int**` and `**float**` function.

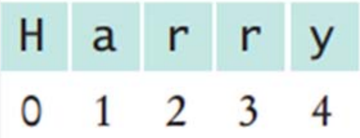
In [ ]:

```
id = int("1729")
price = float("17.29")
print("id is", id, " and price is", price)
```

Strings and Characters

- Strings are sequences of **Unicode** characters.

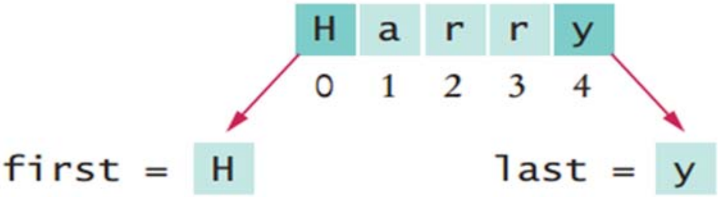
- Individual characters of a string can be accessed based on their position in the string
  - The position is called the **index** of the character.
  - The **index** starts from position 0, followed by 1 for the second character, ... and so on.

- 

  `name = "Harry"`

In [ ]:

```
name = "Harry"
first = name[0]
last = name[4]
```



- The index value must be within the valid range of character positions
  - 0 .. `len(name)-1`
- otherwise, an "index out of range" exception will be generated at run time.

Student Activity

- What are the results of the following statements

```
In [ ]:
 string = "Py"
 string = string + "thon"
```

```
In [ ]:
 print(string)
```

print ("Please" + " enter your name: ")

```
In [ ]:
 print("Please" +
       " enter your name: ")
```

- What is the result of the following statements

```
In [ ]:
 team = str(49) + "ers"
```

```
In [ ]:
 print("team = ", team)
```

```
In [ ]:
 greeting = "H & S"
 n = len(greeting)
```

```
In [ ]:
 print("n = ", n)
```

```
In [ ]:
 string = "Harry"
 n = len(string)
 mystery = string[0] + string[n - 1]
```

```
In [ ]:
 print(mystery)
```

2.5 Input and Output

- Asking the user to provide input values makes programs more flexible.
    - As opposed to having fixed values.

- For example, You will have to change the values of `first` and `second` in the program below every time you would like to use different values.

In [1]:

```
##
#  This program prints a pair of initials.
#

# Set the names of the couple.
first = "Rodolfo"
second = "Sally"

# Compute and display the initials.
initials = first[0] + "&" + second[0]
print(initials)
```

```
R&S
```

- When a program asks for user input, it should first print a message (called a **prompt**) that tells the user which input is expected.
- In Python, displaying a prompt and reading the keyboard input is combined in one operation.

In [ ]:

```
##
#  This program obtains two names from the user and prints a pair of initials
.
#

# Obtain the two names from the user.
first = input("Enter your first name: ")
second = input("Enter your significant other's first name: ")

# Compute and display the initials.
initials = first[0] + "&" + second[0]
print(initials)
```

- Note that the output of the **`input`** function is always a **string**.

Reading Numerical Input

- What if we need to read a numerical input?

- Use the string conversion functions **int** and **float** on the output string

In [2]:

```
userInput = input("Please enter the number of bottles: ")
numberOfBottles = int(userInput)
bottleVolume = float(input("Enter the volume of each bottle: ")) # preferred style
print("The number of bottles = ", numberOfBottles, " and the bottle volume = ", bottleVolume)
```

```
Please enter the number of bottles: 12
Enter the volume of each bottle: 3.2
The number of bottles =  12  and the bottle volume =  3.2
```

Formatted Output

Formatting Floating Point Values

- When you print the result of a computation, you often want to control its appearance.

Instead of	Would Like to Print
Price per liter: 1.215962441314554	Price per liter: 1.22

- We can do that through the **string format operator %**

- The following command displays the price with two digits after the decimal point:

In [3]:

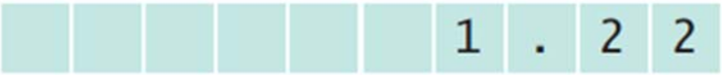
```
price = 1.215962441314554
print("%.2f" % price)
```

```
1.22
```

- You can also specify a field width (the total number of characters, including spaces)

In [ ]:

```
price = 1.215962441314554
print("%7.2f" % price)
```



						1	.	2	2

- `%7.2f` is called a **format specifier**.
- See what happens when you play with the values of the **format specifier**.

Formatting Integer and String Values

- Use **%d** for integer values

```
In [ ]:
 numberOfBottles = 106
 print("%d" % numberOfBottles)
```

- Use **%s** for string values

```
In [ ]:
 title2="Price:"
 print("%-10s" % title2)
```

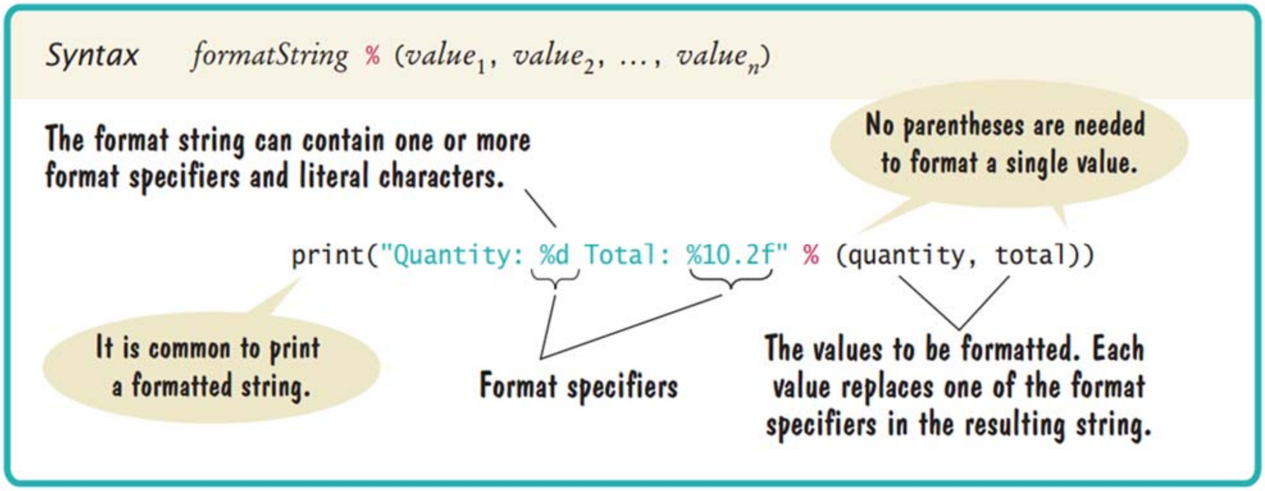
Multiple Format Specifiers

- One can have more than one format specifier in the **format string**
- In this case, the variables to the right of the string format **operator %** need to be included between parentheses and separated by commas.

```
In [ ]:
 quantity = 203
 price = 183.4
 title1 = "Quantity:"
 title2 = "Price:"
 print("%10s %10d" % (title1, quantity))
 print("%10s %10.2f" % (title2, price))
```

- You can play with different values and see what happens to the output
    - print("%-10s %10d" % (title1, quantity))
    - print("%-10s %10.2f" % (title2, price)) # Strings are left aligned, numbers are right aligned
    -
    - print("%10s %-10d" % (title1, quantity)) # Strings are right aligned, numbers are left aligned
    - print("%10s %-10.2f" % (title2, price))
    -
    - print("%-10s %-10d" % (title1, quantity)) # Strings and numbers are left aligned
    - print("%-10s %-10.2f" % (title2, price))

String Format Operator



Syntax formatString % ($value_1$, $value_2$, …, $value_n$)

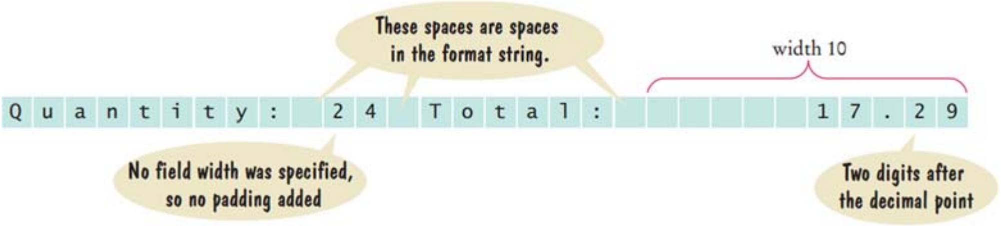
```
print("Quantity: %d Total: %10.2f" % (quantity, total))
```

The format string can contain one or more format specifiers and literal characters.
No parentheses are needed to format a single value.
It is common to print a formatted string.
Format specifiers
The values to be formatted. Each value replaces one of the format specifiers in the resulting string.

- The following statement

In [ ]:

```
quantity = 24
total = 17.29
print("Quantity: %d Total: %10.2f" % (quantity, total))
```



```
Quantity: 24 Total:      17.29
```

These spaces are spaces in the format string.
width 10
No field width was specified, so no padding added
Two digits after the decimal point

- produces

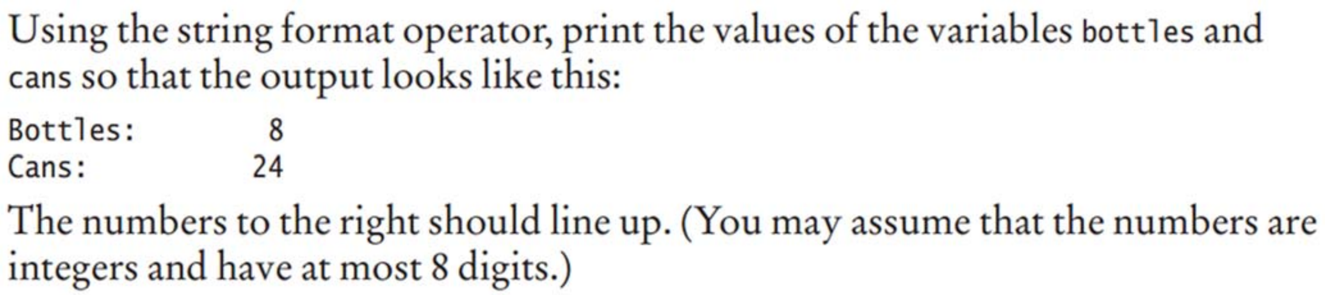
Student Activity

- What is problematic about the following statement sequence?

In [ ]:

```
userInput = input("Please enter the number of cans")
cans = int(userInput)
```

Student Activity



Using the string format operator, print the values of the variables bottles and cans so that the output looks like this:

```
Bottles:        8
Cans:          24
```

The numbers to the right should line up. (You may assume that the numbers are integers and have at most 8 digits.)

In [ ]:

```
# To Print Bottles and Cans
bottles = 8
cans = 24
## Insert your solution here
```

In [ ]:

```
# Different solutions:
print("Bottles: %8d" % bottles)
print("Cans:    %8d" % cans)

print("Bottles: %8d" % bottles)
print("Cans: %11d" % cans)

print("%-8s %8d" % ("Bottles:", bottles))
print("%-8s %8d" % ("Cans:", cans))
```