2.1 Variables and assignments

Remembering a value

Here's a variation on a children's riddle.

PARTICIPATION ACTIVITY 2.1.1: People on bus.

For each step, keep track of the current number of people by typing in the num_people box (it's editable).

Start

```
You are driving a bus.
The bus starts with 5 people.
```

num_people: 5

1	2	3	4	5

Check Next

The real riddle's ending question is actually, "What is the bus driver's name?" The subject usually says, "How should I know?" The riddler then says, "I started with YOU are driving a bus."

The box above serves the same purpose as a *variable* in a program, introduced below.

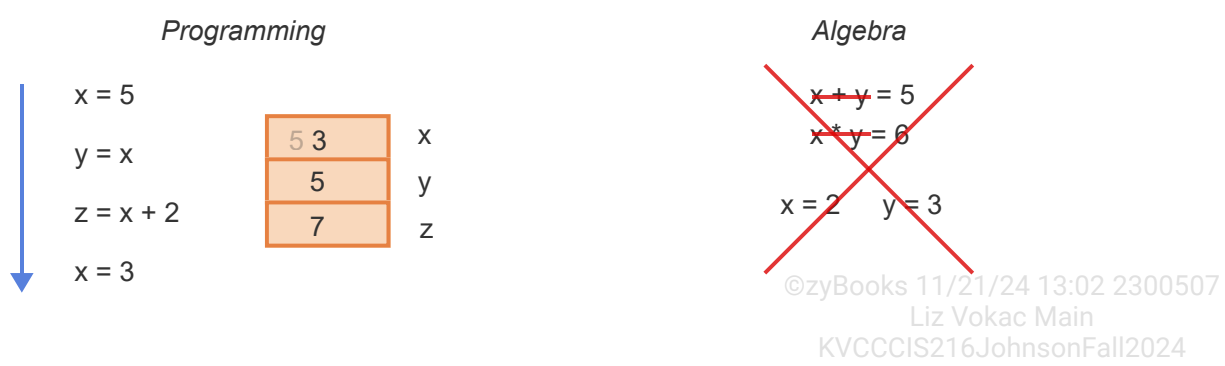
Variables and assignments

In a program, a ***variable*** is a named item, such as x or num_people, that holds a value.

An ***assignment statement*** assigns a variable with a value, such as x = 5. That statement means x is assigned with 5, and x keeps that value during subsequent statements, until x is assigned again.

An assignment statement's left side must be a variable. The right side can be an expression, so a statement may be x = 5, y = x, or z = x + 2. The 5, x, and x + 2 are each an expression that evaluates to a value.

PARTICIPATION ACTIVITY 2.1.2: Variables and assignments.



Animation content:

Static figure:
Begin assignment statements:
Under header programming: x = 5, y = x, z = x + 2, x = 3
Under header algebra: x + y = 5, x * y = 6, x = 2, y = 3
End assignment statements.
The equations labeled as 'Algebra' are crossed out.

Step 1: In programming, a variable is a place to hold a value. Here, variables x, y, and z are depicted graphically as boxes.
The variables x, y, and z are shown next to their respective boxes.

Step 2: An assignment statement assigns the left-side variable with the right-side expression's value. x = 5 assigns x with 5.
5 is moved inside the box corresponding to the variable x, and x is assigned to 5.

Step 3: y = x assigns y with x's value, which presently is 5. z = x + 2 assigns z with x's present value plus 2, so 5 + 2 or 7.
x is replaced with the value stored in the box labeled x. Y = 5 is displayed as a result, and 5 is moved into the box corresponding to the variable y. Next, x is replaced again by 5, and z = 5 + 2 is displayed. 7 is moved into the box corresponding to the variable z.

Step 4: A subsequent x = 3 statement assigns x with 3. x's former value of 5 is overwritten and thus lost. Note that the values held in y and z are unaffected, remaining as 5 and 7.
The value corresponding to the box labeled x is changed from 5 to 3.

Step 5: In algebra, an equation means "the item on the left always equals the item on the right." So for x + y = 5 and x * y = 6, one can determine x = 2 and y = 3.
x + y =5 and x * y = 6 are displayed. Their solutions are shown as x = 2 and y = 3.

Step 6: Assignment statements look similar but have different meanings. The left side must be one variable.
x + y = 5 and x * y = 6 are crossed out, leaving just x = 2 and y = 3 readable.

Step 7: The = isn't "equals," but is an action that PUTS a value into the variable. Assignment statements only make sense when executed in sequence.
All of the algebraic expressions are crossed out. An arrow indicated the following order that the programming assignments must be executed in: x = 5, y = x, z = x + 2, x = 3

Animation captions:

1. In programming, a variable is a place to hold a value. Here, variables x, y, and z are depicted graphically as boxes.
2. An assignment statement assigns the left-side variable with the right-side expression's value. x = 5 assigns x with 5.
3. y = x assigns y with x's value, which presently is 5. z = x + 2 assigns z with x's present value plus 2, so 5 + 2 or 7.
4. A subsequent x = 3 statement assigns x with 3. x's former value of 5 is overwritten and thus lost. Note that the values held in y and z are unaffected, remaining as 5 and 7.
5. In algebra, an equation means "the item on the left always equals the item on the right." So for x + y = 5 and x * y = 6, one can determine x = 2 and y = 3.
6. Assignment statements look similar but have different meanings. The left side must be one variable.
7. The = isn't "equals," but is an action that PUTS a value into the variable. Assignment statements only make sense when executed in sequence.

= is not equals

In programming, = is an assignment of a left-side variable with a right-side value. = is NOT equality, as in mathematics. Thus, x = 5 is read as "x is assigned with 5" and not as "x equals 5." When a programmer sees x = 5, that programmer might think of a value put into a box.

PARTICIPATION ACTIVITY 2.1.3: Valid assignment statements.

Indicate which assignment statements are valid.

1) x = 1

- ○ Valid
- ○ Invalid

2) x = y

- Valid
- Invalid

3) x = y + 2

- Valid
- Invalid

4) x + 1 = 3

- Valid
- Invalid

5) x + y = y + x

- Valid
- Invalid

PARTICIPATION ACTIVITY 2.1.4: Variables and assignment statements.

Given variables x, y, and z:

1) x = 9
y = x + 1
What is y?

Check Show answer

2) x = 9
y = x + 1
What is x?

Check Show answer

3) x = 9
y = x + 1
x = 5
What is y?

Check Show answer

PARTICIPATION ACTIVITY 2.1.5: Trace the variable value.

Select the correct values for x, y, and z after the following statements execute.

Start

```
x = 8
y = 3
z = 0
x = 4
y = 7
z = 8
x = 2
```

x is

2	8	4

y is

3	7	8

z is

5	0	8

1 2 3 4

Check Next

CHALLENGE ACTIVITY 2.1.1: Enter the output of the variable assignments.

Note: Each `print()` includes a newline, so create a newline after your output by pressing *Enter* or *Return* on your keyboard.

566436.4601014.qx3zqy7

Start

Type the program's output

```
x = 9
y = 4

print(x, y)
```

9 4

1 2 3 4 5

Check Next

Assignments with variable on left and right

Because in programming = means assignment, a variable may appear on both the left and right as in x = x + 1. If x was originally 6, x is assigned with 6 + 1, or 7. The statement overwrites the original 6 in x.

Increasing a variable's value by 1, as in x = x + 1, is common and known as ***incrementing*** the variable.

PARTICIPATION ACTIVITY 2.1.6: A variable may appear on the left side and right side of an assignment statement.

```
x = 1
x = x * 20
x = x * 20
print ('A person with measles may cause', x)
print ('people to be infected in two weeks.')
```

1 20 400 x

A person with measles may cause 400 people to be infected in two weeks.

Animation content:

Static figure:
Begin pseudocode:
x = 1
x = x * 20
x = x * 20
print ('A person with measles may cause', x)
print ('people to be infected in two weeks.')
End pseudocode.
"A person with measles may cause 400 people to be infected in two weeks." is printed to the screen.

Step 1: A variable may appear on both sides of an assignment statement. After x = 1, then x = x * 20 assigns x with 1 * 20 or 20, overwriting x's previous 1.
X is set to 1. Then, x = x * 20 executes and x is set to 20. The previous value of x = 1 is grayed out.

Step 2: Another x = x * 20 assigns x with 20 * 20 or 400, which overwrites x's previous 20.
x = x * 20 is executed and x is set to 400. The previous value of x = 20 is grayed out.

Step 3: Only the latest value is held in x.
The following pseudocode executes:
print ('A person with measles may cause', x)
print ('people to be infected in two weeks.')
"A person with measles may cause 400 people to be infected in two weeks." is printed to the screen.

Animation captions:

1. A variable may appear on both sides of an assignment statement. After x = 1, then x = x * 20 assigns x with 1 * 20 or 20, overwriting x's previous 1.
2. Another x = x * 20 assigns x with 20 * 20 or 400, which overwrites x's previous 20.
3. Only the latest value is held in x.

PARTICIPATION ACTIVITY 2.1.7: Variable on both sides.

Indicate the value of x after the statements execute.

1) x = 5
x = x + 7

Check Show answer

2) x = 2
y = 3

x = x * y
x = x * y

Check Show answer

3) y = 30
x = y + 2
x = x + 1

Check Show answer

4) Complete this statement to increment y:

y = _____

Check Show answer

CHALLENGE ACTIVITY 2.1.2: Variables and assignments.

566436.4601014.qx3zqy7

Start

Assign variable num_jackals with 85.

The output is: 85

Learn how our autograder works

```
1
2 ''' Your code goes here '''
3
4 print(num_jackals)
```

1 2 3

Check Next level

©zyBooks 11/21/24 13:02 2300507
Liz Vokac Main
KVCCCIS216JohnsonFall2024

2.2 Identifiers

Rules for identifiers

A programmer gives names to various items, such as variables (and also functions, described later). For example, `x = 5` uses the name "x" to refer to the value 5. An ***identifier***, also called a ***name***, is a sequence of letters (a-z, A-Z), ***underscores*** (_), and digits (0–9), and must start with a letter or an underscore.

Python is ***case sensitive***, meaning uppercase and lowercase letters differ. Ex: "Cat" and "cat" are different. The following are valid names: c, cat, Cat, n1m1, short1, and _hello. The following are invalid names: 42c (doesn't start with a letter or underscore), hi there (has a space), and cat$ (has a symbol other than a letter, digit, or underscore).

Names that start and end with double underscores (for example, __init__) are allowed but should be avoided because Python has special usages for double underscore names, explained elsewhere. A good variable name should describe the purpose of the variable, such as "temperature" or "age," rather than just "t" or "A".

Certain words like "and" or "True" cannot be used as names. ***Reserved words***, or ***keywords***, are words that are part of the language and cannot be used as a programmer-defined name. Many language editors will automatically color a program's reserved words. A list of reserved words appears at the end of this section.

PARTICIPATION ACTIVITY 2.2.1: Valid names.

Which of the following are valid names?

1) numCars
   - Valid
   - Invalid

2) num_cars1
   - Valid
   - Invalid

3) _num_cars
   - Valid
   - Invalid

4) ___numcars2
   - Valid
   - Invalid

5) num cars
   - Valid

- Invalid

6) 3rd_place

- Valid
- Invalid

7) third_place_

- Valid
- Invalid

8) third_place!

- Valid
- Invalid

9) output

- Valid
- Invalid

10) return_copy

- Valid
- Invalid

Style guidelines for identifiers

A good practice when naming variables is to use all lowercase letters and to place underscores between words. This lowercase and underscore convention for naming variables originates from PEP 8, the Python style guide. ***PEP 8*** (Python Enhancement Proposal) outlines the basics of how to write Python code neatly and consistently. Code is read more often than written, so having a consistent variable naming scheme helps to ensure that programmers can understand each other's code.

Programmers should create meaningful names that describe an item's purpose. If a variable will store a person's age, then a name like "age" is better than "a". *A good practice when dealing with scientific or engineering names is to append the unit of measure. Ex: Instead of temperature, use temperature_celsius.* Abbreviations should only be used if widely understandable, as in tv_model or ios_app. While meaningful names are important, very long variable names, such as "average_age_of_a_UCLA_graduate_student," can make subsequent statements too long and thus hard to read, so programmers find a balance between meaningful names and short names. Below are some examples of names that perhaps are less and more meaningful.

Table 2.2.1: Use meaningful variable names.

Purpose	Less meaningful names	More meaningful names
The number of students attending the class	ns num nu	num_students
The size of a television set measured by its diagonal length	sz_tv size	diagonal_tv_size_inches
The ratio of a circle's circumference/diameter	p	pi
The number of jelly beans in a jar, as guessed by a user	guess num njb	num_guessed_jelly_beans user_guess_jelly_beans

A list of reserved keywords in the language are shown below:

Table 2.2.2: Python 3 reserved keywords.

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


Source: http://docs.python.org/3/reference/lexical_analysis.html

PARTICIPATION ACTIVITY 2.2.2: Python 3 name validator.

Use the tool below to test valid and invalid names.

Enter an identifier: _____ **Validate**

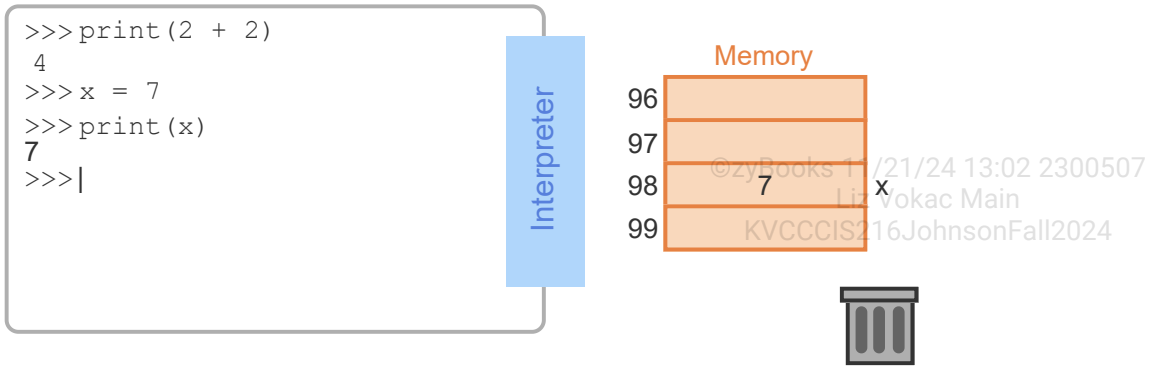
2.3 Experimenting with objects.

Objects

The Python interpreter is a program that runs on a computer, just like an Internet browser or a text editor. Instead of displaying a web page or creating a document, the interpreter runs Python programs. An ***object*** represents a value and is automatically created by the interpreter when executing a line of code. For example, executing `x = 4` creates a new object to represent the value 4. A programmer does not explicitly create objects; instead, the interpreter creates and manipulates objects as needed to run the Python code. Objects are used to represent everything in a Python program, including integers, strings, functions, lists, etc.

The animation below shows some objects being created while executing Python code statements in an interactive Python interpreter. The interpreter assigns an object to a location somewhere in memory automatically.

PARTICIPATION ACTIVITY 2.3.1: Creating new objects.



Animation content:

Static figure:
Begin interpreter code:
print(2 + 2)
4
x = 7
print(x)
7
End interpreter code.
Memory addresses 96 to 99 are shown. Variable x with value 7 is at memory address 98. There is a trash can.

Step 1: The interpreter creates a new object with the value 4. The object is stored somewhere in memory.
print(2 + 2) is run, and 2 + 2 moves to the box labeled interpreter. 2 + 2 changes to 4 and moves to memory address 96.

Step 2: Once 4 is printed, the object is no longer needed and is thrown away.
4 at memory address 96 duplicates and moves to output on the interpreter. 4 at memory address 96 moves into the trash.

Step 3: New object created: "x" references the object stored in address 98.
x = 7 is run, and x is assigned to memory address 98. The value at memory address 98 becomes 7.

Step 4: Objects are retrieved from memory when needed.
print(x) is run, and 7 at memory address 98 duplicates and moves to output on the interpreter.

Animation captions:

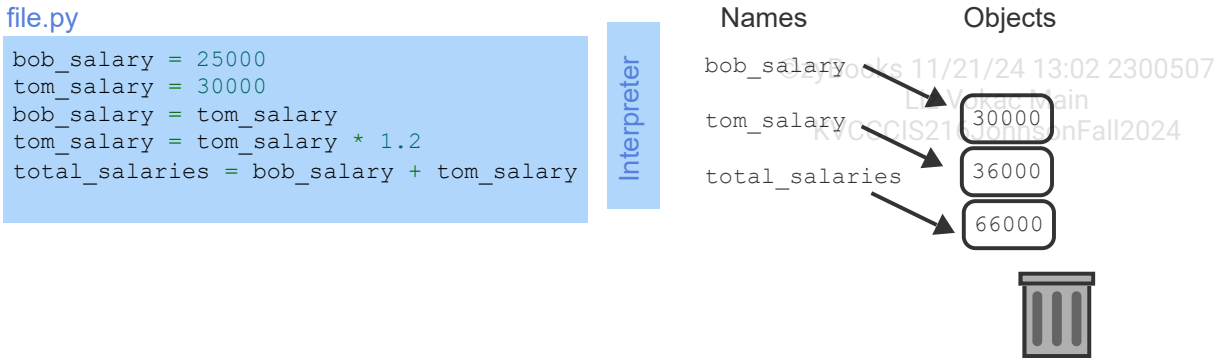
1. The interpreter creates a new object with the value 4. The object is stored somewhere in memory.
2. Once 4 is printed, the object is no longer needed and is thrown away.
3. New object created: "x" references the object stored in address 98.
4. Objects are retrieved from memory when needed.

Above, the interpreter performs an addition of 2 + 2, resulting in the creation of a new object with a value of 4. Once 4 is printed, the object is no longer needed, so the object is automatically deleted from memory and thrown away. Deleting unused objects is an automatic process called ***garbage collection*** that frees memory space

Name binding

Name binding is the process of associating names with interpreter objects. An object can have more than one name bound to it, and every name is always bound to exactly one object. Name binding occurs whenever an assignment statement is executed, as demonstrated below.

PARTICIPATION ACTIVITY 2.3.2: Manipulating variables.



Animation content:

Static figure:
Begin Python code:
bob_salary = 25000
tom_salary = 30000
bob_salary = tom_salary
tom_salary = tom_salary * 1.2
total_salaries = bob_salary + tom_salary
End Python code.
Under header Names: bob_salary, tom_salary, total_salaries
Under header Objects: 30000, 36000, 66000
bob_salary points to 30000, tom_salary points to 36000, and total_salaries points to 66000.
There is an interpreter between the code and memory. There is a trash can.

Step 1: bob_salary object is created by the interpreter.
The line of code, bob_salary = 25000, is highlighted. The line of code duplicates and moves to the interpreter. The interpreter moves bob_salary to header Names and 25000 to header Objects. An arrow points from bob_salary to 25000.

Step 2: tom_salary object is created by the interpreter.
The line of code, tom_salary = 30000, is highlighted. The line of code duplicates and moves to the interpreter. The interpreter moves tom_salary to header Names and 30000 to header Objects. An arrow points from tom_salary to 30000.

Step 3: bob_salary is assigned tom_salary, and the 25000 object is garbage collected.
The line of code, bob_salary = tom_salary is highlighted. The arrow from bob_salary to 25000 now

points from bob_salary to 30000. 25000 moves to the trash.

Step 4: tom_salary is assigned tom_salary * 1.2.
The line of code, tom_salary = tom_salary * 1.2, is highlighted. The line of code duplicates and moves to the interpreter. The Object 30000 duplicates and moves to the interpreter. 30000 updates to 36000 and moves back to Objects. The arrow from tom_salary to 30000 now points from tom_salary to 36000.

Step 5: total_salaries object is created by the interpreter and is assigned bob_salary + tom_salary.
The line of code, total_salaries = bob_salary + tom_salary, is highlighted. The line of code duplicates and moves to the interpreter. The Objects 30000 and 36000 duplicate and move to the interpreter. 66000 is created. total_salaries and 66000 move to Names and Object respectively. An arrow points from total_salaries to 66000.

Animation captions:

1. bob_salary object is created by the interpreter.
2. tom_salary object is created by the interpreter.
3. bob_salary is assigned tom_salary, and the 25000 object is garbage collected.
4. tom_salary is assigned tom_salary * 1.2.
5. total_salaries object is created by the interpreter and is assigned bob_salary + tom_salary.

Properties of objects

Each Python object has three defining properties: value, type, and identity.

1. ***Value***: A value such as "20", "abcdef", or "55".
2. ***Type***: The type of the object, such as integer or string.
3. ***Identity***: A unique identifier that describes the object.

The *value* of an object is the data associated with the object. For example, evaluating the expression 2 + 2 creates a new object whose value is 4. The value of an object is examined by printing that object.

Figure 2.3.1: Printing displays an object's value.


```
x = 2 + 2     # Create a new object with a value of 4, referenced by
'x'.
print(x)      # Print the value of the object.
print(5)
```

```
4
5
```

The *type* of an object determines the object's supported behavior. For example, integers can be added and multiplied, while strings can be appended with additional text or concatenated together. An object's type never changes once created. The built-in function ***type()*** returns the type of an object.

The type of an object also determines the mutability of an object. ***Mutability*** indicates whether the object's value is allowed to be changed. Integers and strings are ***immutable***; meaning integer and string values cannot be changed. Modifying the values with assignment statements results in new objects being created and the names bound to the new object.

Figure 2.3.2: Using type() to print an object's type.

```
x = 2 + 2            # Create a new object with a value of 4,
referenced by 'x'.
print(type(x))       # Print the type of the object.

print(type('ABC'))   # Create and print the type of a string
object.
```

```
<class
'int'>
<class
'str'>
```

The *identity* of an object is a unique numeric identifier, such as 1, 500, or 505534. Only one object at any time may have a particular identifier. The identity normally refers to the memory address where the object is stored. Python provides a built-in function ***id()*** that gives the value of an object's identity.

Figure 2.3.3: Using id() to print an object's identity.

```
x = 2 + 2            # Create a new object with a value of 4,
referenced by 'x'
print(id(x))         # Print the identity (memory address) of the
x object

print(id('ABC'))     # Create and print the identity of a string
('ABC') object
```

```
1752608
2330312
```


Try 2.3.1: Experimenting with objects.

Full screen

Run the following code and observe the results of type() and id(). Create a new object called "age" that has a value of 19, and print the id and type of the new object.

Run History Tutorial

```
birthday_year = 1986
birthday_month = 'April'
birthday_day = 22

print('birthday_year -->')
print(' value:', birthday_year)
print(' type:', type(birthday_year))
print(' id:', id(birthday_year))

print('\nbirthday_month -->')
print(' value:', birthday_month)
print(' type:', type(birthday_month))
print(' id:', id(birthday_month))

print('\nbirthday_day -->')
print(' value:', birthday_day)
print(' type:', type(birthday_day))
print(' id:', id(birthday_day))

```


CONSOLE

Model Solution Show

PARTICIPATION ACTIVITY 2.3.3: Objects basics.

1) Which built-in function finds the type of an object?

Check Show answer

2) Write an expression that gives the identity of a variable called my_num.

Check Show answer

2.4 Numeric types: Floating-point

Floating-point numbers and scientific notation

A ***floating-point number*** is a real number, like 98.6, 0.0001, or -666.667. The term "floating-point" refers to the decimal point appearing anywhere ("floating") in the number. Thus, ***float*** is a data type for floating-point numbers.

A ***floating-point literal*** is written with the fractional part even if that fraction is 0, as in 1.0, 0.0, or 99.0.

Figure 2.4.1: A program using float-type variables.

The below program reads in a floating-point value from a user and calculates the time to drive and fly the distance. Note the use of the built-in function float() when reading the input to convert the input string into a float.

Note that print handles floating-point numbers straightforwardly.

```
miles = float(input('Enter a distance in miles:
'))
hours_to_fly = miles / 500.0
hours_to_drive = miles / 60.0

print(miles, 'miles would take:')
print(hours_to_fly, 'hours to fly')
print(hours_to_drive, 'hours to drive')
```

```
Enter a distance in miles:
450
450.0 miles would take:
0.9 hours to fly
7.5 hours to drive
...
Enter a distance in miles:
1800
1800.0 miles would take:
3.6 hours to fly
30.0 hours to drive
```


Scientific notation is useful for representing floating-point numbers that are much greater than or much less than 0, such as 6.02×10^{23}. A floating-point literal using ***scientific notation*** is written using an "e" preceding the power-of-10 exponent, as in 6.02e23 to represent 6.02×10^{23}. The "e" stands for exponent. Likewise, 0.001 is 1×10^{-3}, so it can be written as 1.0e-3.

PARTICIPATION ACTIVITY 2.4.1: Scientific notation.

1) Type 1.0e-4 as a floating-point literal but not using scientific notation, with a single digit before and four digits after the decimal point.

Check Show answer

2) Type 7.2e-4 as a floating-point literal without using scientific notation, using a single digit before and five digits after the decimal point.

Check Show answer

3) Type 540,000,000 as a floating-point literal using scientific notation with a single digit before and after the decimal point.

Check Show answer

4) Type 0.000001 as a floating-point literal using scientific notation with a single digit before and after the decimal point.

Check Show answer

5) Type 623.596 as a floating-point literal using scientific notation with a single digit before and five digits after the decimal point.

Check Show answer

zyDE 2.4.1: Energy to mass conversion.

©zyBooks 11/21/24 13:02 2300507
Liz Vokac Main
KVCCCIS216JohnsonFall2024

Albert Einstein's equation $E = mc^2$ is likely the most widely known mathematical formula. The equation describes the mass-energy equivalence, which states that the mass (amount of matter) *m* of a body is directly related to the amount of energy *E* of the body, connected via a constant value c^2, the speed of light squared. The significance of the equation is that matter can be converted to energy, (and theoretically, energy back to matter). The mass-energy equivalence equation can be used to calculate the energy released in nuclear reactions, such as nuclear fission or nuclear fusion, which form the basis of modern technologies like nuclear weapons and nuclear power plants.

The following program reads in a mass in kilograms and prints the amount of energy stored in the mass. Also printed are the equivalent numbers of AA batteries and tons of TNT.

Load default template...

```
c_meters_per_sec = 299792458  # Speed of light (m/s)
joules_per_AA_battery = 4320.5  # Nickel-Cadmium AA batter
joules_per_TNT_ton = 4.184e9

#Read in a floating-point number from the user
mass_kg = float(input())

#Compute E = mc^2.
energy_joules = mass_kg * (c_meters_per_sec**2)  # E = mc^
print('Total energy released:', energy_joules, 'Joules')

#Calculate equivalent number of AA and tons of TNT.
num_AA_batteries = energy_joules / joules_per_AA_battery
num_TNT_tons = energy_joules / joules_per_TNT_ton

print('Which is as much energy as:')
print('  ', num_AA_batteries, 'AA batteries')
print('  ', num_TNT_tons, 'tons of TNT')
```

0.1

Run

Overflow

Float-type objects have a limited range of values that can be represented. For a standard 32-bit installation of Python, the maximum floating-point value is approximately 1.8×10^{308}, and the minimum

floating-point value is 2.3×10^{-308}. Assigning a floating-point value outside of this range generates an ***OverflowError***. ***Overflow*** occurs when a value is too large to be stored in the memory allocated by the interpreter. Ex: The program in the figure below tries to store the value 2.0^{1024}, which causes an overflow error.

In general, floating-point types should be used to represent quantities that are measured, such as distances, temperatures, volumes, and weights. Integer types should be used to represent quantities that are counted, such as numbers of cars, students, cities, hours, and minutes.

Figure 2.4.2: Float can overflow.

```
print('2.0 to the power of 256 =', 2.0**256)
print('2.0 to the power of 512 = ', 2.0**512)
print('2.0 to the power of 1024 = ', 2.0**1024)
```

```
2.0 to the power of 256 = 1.15792089237e+77
2.0 to the power of 512 = 1.34078079299e+154
2.0 to the power of 1024 =
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
OverflowError: (34, 'Result too large')
```

PARTICIPATION ACTIVITY 2.4.2: Floating-point versus integer.

Choose the right type for a variable to represent each item.

1) The number of cars in a parking lot.
   - ○ float
   - ○ int

2) The current temperature in Celsius.
   - ○ float
   - ○ int

3) A person's height in centimeters.
   - ○ float
   - ○ int

4) The number of hairs on a person's head.
   - ○ float
   - ○ int

5) The average number of kids per household.

- ○ float
- ○ int

Manipulating floating-point output

Some floating-point numbers have many digits after the decimal point. Ex: Irrational numbers (Ex: 3.14159265359...) and repeating decimals (Ex: 4.33333333...) have an infinite number of digits after the decimal. By default, most programming languages output at least five digits after the decimal point. But for many simple programs, this level of detail is not necessary. A common approach is to output floating-point numbers with a specific number of digits after the decimal to reduce complexity or produce a certain numerical type (Ex: Representing currency with two digits after the decimal). The syntax for outputting the float myFloat with two digits after the decimal point is `print(f'{myFloat:.2f}')`

When outputting a certain number of digits after the decimal using print(), Python rounds the last output digit, but the floating-point value remains the same. Manipulating how numbers are output is discussed in detail elsewhere.

PARTICIPATION ACTIVITY 2.4.3: Reducing the output of Pi.

```
import math

print('Default output of Pi:',  math.pi)
print('Pi reduced to 4 digits after the decimal:', end=' ')
print(f'{math.pi:.4f}')
```

```
Default output of Pi: 3.141592653589793
Pi reduced to 4 digits after the decimal: 3.1416
```

Animation content:

Static figure:
Begin Python code:
import math

print('Default output of Pi:', math.pi)
print('Pi reduced to 4 digits after the decimal:', end=' ')

print(f'{math.pi:.4f}')
End Python code.

The output for the code is displayed in an output box. This is the two-line output:
Default output of pi: 3.141592653589793
Pi reduced to 4 digits after the decimal: 3.1416
End output.

Step 1: The mathematical constant Pi (π) is irrational. Pi is a floating-point number whose digits after the decimal point are infinite and non-repeating. The math module defines the constant pi with the value of Pi.
The output display is empty. In the first line of code, "math.pi" is highlighted. The value "3.141592653589793..." appears above the highlighted variable name "math.pi".

Step 2: Though Python does not attempt to output the full value of Pi, Python outputs 15 digits after the decimal by default.
The line of code, "print('Default output of Pi:', math.pi)", is highlighted. Then, the text "Default output of Pi: 3.141592653589793" appears in the output display.

Step 3: print(f'{math.pi:.4f}') outputs Pi to only four digits after the decimal. The last digit is rounded up in the output, but the value of Pi remains the same.
The line of code, "print('Default output of Pi:', math.pi)", is no longer highlighted and the lines of code, "print('Pi reduced to 4 digits after the decimal:', end=' ')" and "print(f'{math.pi:.4f}')". The text "Pi reduced to 4 digits after the decimal: 3.1416" then appears in the output.

Animation captions:

1. The mathematical constant Pi (π) is irrational. Pi is a floating-point number whose digits after the decimal point are infinite and non-repeating. The math module defines the constant pi with the value of Pi.
2. Though Python does not attempt to output the full value of Pi, Python outputs 15 digits after the decimal by default.
3. print(f'{math.pi:.4f}') outputs Pi to only four digits after the decimal. The last digit is rounded up in the output, but the value of Pi remains the same.

PARTICIPATION ACTIVITY 2.4.4: Reducing floating-point output.

1) Which of the following arguments completes print() to output two digits after the decimal point?

```
print(f'{(7.0 / 3.0)_____)
```

- ○ .2f}'

- 2:f}'
- :.2f}'

2) What is output by `print(f'{0.125:.1f}')`?

- 0
- 0.1
- 0.13

3) What is output by `print(f'{9.1357:.3f}')`?

- 9.136
- 9.135
- 9.14

CHALLENGE ACTIVITY 2.4.1: Gallons of paint needed to paint walls.

The number of gallons of paint needed to cover a wall is equal to the wall's total area divided by 350.0. The program reads wall_area from input, representing a wall's total area. Then, the program computes num_gallons as wall_area / 350.0.

Complete the program to output the number of gallons of paint needed to cover the wall, rounded to five digits after the decimal point.

▶ **Click here for examples**

Note: `print(f'{my_float:.Xf}')` outputs my_float to X digits after the decimal point.

See How to Use zyBooks for info on how our automated program grader works.

566436.4601014.qx3zqy7

```
1 # Assign wall_area with a float read from input
2 wall_area = float(input())
3
4 # Compute num_gallons
5 num_gallons = wall_area / 350.0
6
7 ''' Your solution goes here '''
8
```

Run

View your last submission

CHALLENGE ACTIVITY 2.4.2: Numeric types: Floating-point.

566436.4601014.qx3zqy7

Start

One minute equals 60 seconds. The following program intends to read a floating-point value from input minutes, but the code contains an error when reading from input. Find and fix the error.

Ex: If the input is `3.7`, then the output is:

```
0.062 minutes
```

```
# Modify the following line of code
length_secs = int(input())

length_mins = length_secs / 60

print(f'{length_mins:.3f} minutes')
```

1 2 3

Check Next level

2.5 Arithmetic expressions

Basics

An ***expression*** is a combination of items, like variables, literals, operators, and parentheses, that evaluates to a value, like 2 * (x + 1). A common place where expressions are used is on the right side of an assignment statement, as in y = 2 * (x + 1).

A ***literal*** is a specific value in code like 2. An ***operator*** is a symbol that performs a built-in calculation, like +, which performs addition. Common programming operators are shown below.

Table 2.5.1: Arithmetic operators.

Arithmetic operator	Description
+	The ***addition*** operator is **+**, as in x + y.
-	The ***subtraction*** operator is **-**, as in x - y. Also, the - operator is for ***negation***, as in -x + y, or x + -y.
*	The ***multiplication*** operator is ***, as in x * y.
/	The ***division*** operator is **/**, as in x / y.
**	The ***exponent*** operator is ****, as in x ** y (x to the power of y).

PARTICIPATION ACTIVITY 2.5.1: Expressions.

Indicate which are valid expressions. x and y are variables.

1) x + 1

- Valid
- Not valid

2) 2 * (x - y)

- Valid
- Not valid

3) x

- Valid
- Not valid

4) 2

- Valid
- Not valid

5) 2x

- Valid
- Not valid

6) 2 + (xy)

- Valid
- Not valid

7) y = x + 1

- Valid
- Not valid

PARTICIPATION ACTIVITY 2.5.2: Capturing behavior with an expression.

Does the expression correctly capture the intended behavior?

1) 6 plus num_items:

```
6 + num_items
```

- Yes
- No

2) 6 times num_items:

```
6 x num_items
```

- Yes
- No

3) total_days divided by 12:

```
total_days / 12
```

- Yes
- No

4) 5 times t:

```
5t
```

- Yes
- No

5) The negative of user_val:


```
-user_val
```

- Yes
- No

6) n factorial

```
n!
```

- Yes
- No

Evaluation of expressions

An expression ***evaluates*** to a value, which replaces the expression. Ex: If x is 5, then x + 1 evaluates to 6, and y = x + 1 assigns y with 6.

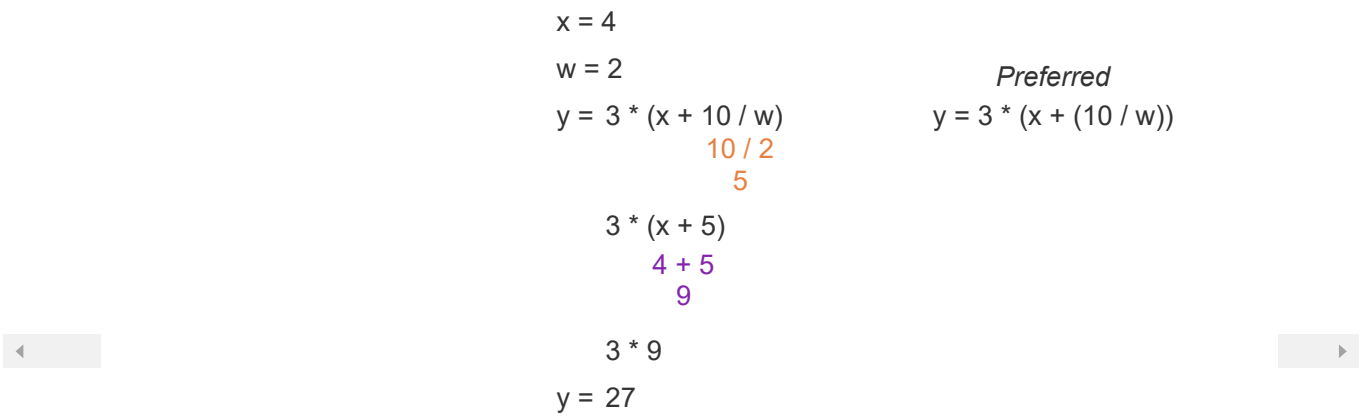
An expression is evaluated using the order of standard mathematics, and is also known in programming as ***precedence rules***, listed below.

Table 2.5.2: Precedence rules for arithmetic operators.

Operator/Convention	Description	Explanation
( )	Items within parentheses are evaluated first.	In 2 * (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2.
exponent **	** used for exponent is next.	In x**y * 3, x to the power of y is computed first, with the results then multiplied by 3.
unary -	- used for negation (unary minus) is next.	In 2 * -x, the -x is computed first, with the result then multiplied by 2.

*** / %**	Next to be evaluated are *, /, and %, having equal precedence.	(% is discussed elsewhere.)
+ -	Finally come + and - with equal precedence.	In y = 3 + 2 * x, the 2 * x is evaluated first, with the result then added to 3, because * has higher precedence than +. Spacing doesn't matter because y = 3+2 * x would still evaluate 2 * x first.
left-to-right	If more than one operator of equal precedence could be evaluated, evaluation occurs left to right. Note: The ** operator is evaluated from right-to-left.	In y = x * 2 / 3, the x * 2 is first evaluated, with the result then divided by 3.

PARTICIPATION ACTIVITY 2.5.3: Evaluating expressions.



Animation content:

Static figure:
Begin math:
x = 4
w = 2
y = 3 * (x + 10 / w)
10 / 2

5
3 * (x + 5)
4 + 5
9
3 * 9
y = 27
End math.
Under header Preferred: y = 3 * (x + (10 / w))
End figure.

Step 1: An expression such as 3 * (x + 10 / w) evaluates to a value, using precedence rules. Items within parentheses come first, and / comes before +, yielding 3 * (x + 5).

Shown are three equations: "x = 4", "w = 2", "y = 3 * (x + 10 / w)". Superimposed on the last equation, atop "10 / w", is the expression "10 / 2". "10 / 2" then descends from atop the equation to below the equation. The number 5 then appears below "10 / 2" and the equation "3 * (x + 5)" appears below the number 5.

Step 2: Evaluation finishes inside the parentheses: 3 * (x + 5) becomes 3 * 9.
Below "3 * (x + 5)" appears "4 + 5", below which "9" appears. "3 * 9" then appears below the 9.

Step 3: Thus, the original expression evaluates to 3 * 9 or 27. That value replaces the expression. So y = 3 * (x + 10 / w) becomes y = 27, so y is assigned with 27.
"y = 27" appears below "3 * 9".

Step 4: Many programmers prefer to use parentheses to make order of evaluation more clear when the order is not obvious.
"y = 3 * (x + 10 / w)" appears to the side, "y = 3 * (x + (10 / w))" under the header, Preffered.

Animation captions:

1. An expression such as 3 * (x + 10 / w) evaluates to a value, using precedence rules. Items within parentheses come first, and / comes before +, yielding 3 * (x + 5).
2. Evaluation finishes inside the parentheses: 3 * (x + 5) becomes 3 * 9.
3. Thus, the original expression evaluates to 3 * 9 or 27. That value replaces the expression. So y = 3 * (x + 10 / w) becomes y = 27, so y is assigned with 27.
4. Many programmers prefer to use parentheses to make order of evaluation more clear when the order is not obvious.

PARTICIPATION ACTIVITY 2.5.4: Evaluating expressions and precedence rules.

Select the expression in which parentheses match the evaluation order of the original expression.

1) y + 2 * z

- (y + 2) * z
- y + (2 * z)

2) z / 2 - x

- (z / 2) - x
- z / (2 - x)

3) x * y * z

- (x * y) * z
- x * (y * z)

4) x + 1 * y/2

- ((x + 1) * y) / 2
- x + ((1 * y) / 2)
- x + (1 * (y / 2))

5) x / 2 + y / 2

- ((x / 2) + y) / 2
- (x / 2) + (y / 2)

6) What is total_count after executing the following?

```
num_items = 5
total_count = 1 + (2 *
num_items)  * 4
```

- 44
- 41

CHALLENGE ACTIVITY 2.5.1: Precedence rules for arithmetic operators.

566436.4601014.qx3zqy7

Start

a - f * 30

Which operator is evaluated first? Select one

Check Next

Using parentheses to make the order of evaluation explicit

*A common error is to omit parentheses and assume an incorrect order of evaluation, leading to a bug. Ex: If x is 3, then 5 * x+1 might appear to evaluate as 5 * (3+1) or 20, but actually evaluates as (5 * 3) + 1 or 16 (spacing doesn't matter). Good practice is to use parentheses to make order of evaluation explicit, rather than relying on precedence rules, as in: y = (m * x) + b, unless order doesn't matter as in x + y + z.*

Example: Calorie expenditure

A website lists the calories expended by men and women during exercise as follows (source):

Men: Calories = [(Age × 0.2017) + (Weight × 0.09036) + (Heart Rate × 0.6309) — 55.0969] × Time / 4.184

Women: Calories = [(Age × 0.074) — (Weight × 0.05741) + (Heart Rate × 0.4472) — 20.4022] × Time / 4.184

Below are those expressions written using programming notation:

```
calories_man = ( (age_years * 0.2017) + (weight_pounds * 0.09036) + (heart_bpm * 0.6309) - 55.0969 ) * time_minutes / 4.184

calories_woman = ( (age_years * 0.074) - (weight_pounds * 0.05741) + (heart_bpm * 0.4472) - 20.4022 ) * time_minutes/ 4.184
```


PARTICIPATION ACTIVITY 2.5.5: Converting a formatted expression to a program expression.

Consider the example above. Match the changes that were made.

If unable to drag and drop, refresh the page.

× [] Spaces in variable names —

	Replaced by ( )
	Underscores
	-
	*

Reset

CHALLENGE ACTIVITY 2.5.2: Calculate the values of the integer expressions.

566436.4601014.qx3zqy7

Start

Type the program's output

```
x = 5
y = x + 3

print(y)
```

8

1 2 3 4 5

Check Next

2.6 Python expressions

Arithmetic expressions

Below is a simple program that includes an expression involving integers.

Figure 2.6.1: Expression example: Leasing cost.

```
""" Computes the total cost of leasing a car given the
down payment,
    monthly rate, and number of months """

down_payment = int(input('Enter down payment: '))
payment_per_month = int(input('Enter monthly payment:
'))
num_months = int(input('Enter number of months: '))

total_cost = down_payment + (payment_per_month *
num_months)

print (f'Total cost: ${total_cost:.2f}')
```

```
Enter down payment:
500
Enter monthly
payment: 300
Enter number of
months: 60
Total cost: $18500.00
```


PARTICIPATION ACTIVITY 2.6.1: Simple program with an arithmetic expression.

Consider the example above.

1) Would removing the parentheses as shown below yield the same result?

```
down_payment + payment_per_month
* num_months
```

- Yes
- No

2) Would using two assignment statements as below have yielded the same result?

```
all_months_cost =
payment_per_month * num_months
total_cost = down_payment +
all_months_cost
```

- Yes
- No

Style: Single space around operators

A good practice is to include a single space around operators for readability, as in num_items + 2, rather than num_items+2. An exception is minus used as negative, as in: x_coordinate = -y_coordinate. Minus (-) used as negative is known as ***unary minus***.

PARTICIPATION ACTIVITY 2.6.2: Single space around operators.

Retype each statement to follow the good practice of a single space around operators.

Note: If an answer is marked wrong, something differs in the spacing, spelling, capitalization, etc. This activity emphasizes the importance of such details.

1) `houses_city = houses_block *10`

Check Show answer

2) `total = num1+num2+2`

Check Show answer

3) `num_balls=num_balls+1`

Check Show answer

4) `num_entries = (user_val+1)*2`

Check Show answer

Compound operators

Special operators called ***compound operators*** provide a shorthand way to update a variable, such as age **+=** 1 being shorthand for age = age + 1. Other compound operators include **-=**, ***=**, **/=**, and **%=**.

Table 2.6.1: Compound operators.

Compound operator	Expression with compound operator	Equivalent expression
Addition assignment	`age += 1`	`age = age + 1`
Subtraction assignment	`age -= 1`	`age = age - 1`
Multiplication assignment	`age *= 1`	`age = age * 1`
Division assignment	`age /= 1`	`age = age / 1`
Modulo (operator discussed elsewhere) assignment	`age %= 1`	`age = age % 1`

PARTICIPATION ACTIVITY 2.6.3: Compound operators.

1) num_atoms is initially 7. What is num_atoms after:
`num_atoms += 5`?

Check Show answer

2) num_atoms is initially 7. What is num_atoms after:
`num_atoms *= 2`?

Check Show answer

3) Rewrite the statement using a compound operator, or type: Not possible
`car_count = car_count / 2`

Check Show answer

4) Rewrite the statement using a compound operator, or type: Not possible

```
num_items = box_count + 1
```


Check Show answer

No commas allowed

Commas are not allowed in an integer literal. So 1,333,555 is written as 1333555.

PARTICIPATION ACTIVITY 2.6.4: Assigning an integer literal.

1) The following code correctly assigns num_years with an integer value of 2 billion.

```
num_years = 2,000,000,000
```

- True
- False

CHALLENGE ACTIVITY 2.6.1: Python expressions.

566436.4601014.qx3zqy7

Start

The area of a circle is calculated using the formula $Area = (r^2) * \pi$, where r is the circle's radius.

Given variable radius_of_circle read from input and constant PI, compute the area of a circle and assigr area.

▶ **Click here for example**

```
PI = 3.14159

radius_of_circle = float(input())

''' Your code goes here '''

print(f'Circle area: {area_of_circle:.2f}')
```

1 2

Check Next level

2.7 Division and modulo

Division: Integer rounding

The division operator / performs division and returns a floating-point number. Ex:

- 20 / 10 is 2.0.
- 50 / 50 is 1.0.
- 5 / 10 is 0.5.

The floor division operator // can be used to round down the result of a floating-point division to the closest smaller whole number value. The resulting value is an integer type if both operands are integers; if either operand is a float, then a float is returned:

- 20 // 10 is 2.
- 50 // 50 is 1.
- 5 // 10 is 0. (5 = 10 * 0 + 5. So, the result is 0 and the remainder 5 is thrown away.)
- 5.0 // 2 is 2.0.

For division, the second operand of / or // must never be 0, because division by 0 is mathematically undefined.

PARTICIPATION ACTIVITY 2.7.1: Division and floor division.

Determine the result. Type "Error" if the program would terminate due to division by 0. If the answer is a floating-point number, answer in the form #.#, even if the answer is a whole

number.

1) `12 / 4`

Check Show answer

2) `5 / 10`

Check Show answer

3) `5.0 // 2`

Check Show answer

4) `100 / 0`

Check Show answer

Modulo (%)

The basic arithmetic operators include not just +, -, *, /, //, but also %. The ***modulo operator*** (**%**) evaluates the remainder of the division of two integer operands. Ex: 23 % 10 is 3.

Examples:

- 9 % 5 is 4. Reason: Since 9 = 5 * 1 + 4, the floor division 9 // 5 results in 1, and the remainder is 4.
- 70 % 7 is 0. Reason: 70 // 7 is 10 with remainder 0.
- 1 % 2 is 1. Reason: 1 // 2 is 0 with remainder 1.

zyDE 2.7.1: Example using expressions: Minutes to hours/minutes.

The program below reads the number of minutes entered by a user. The program then converts the number of minutes to hours and minutes.

Run the program, then modify the code to work in reverse: The user enters two numbers for hours and minutes and the program outputs total minutes.

Load default template...

```
1 minutes = int(input('Enter mi
2 hours = minutes // 60
3 minutes_remaining = minutes %
4
5 print(minutes, 'minute(s) is'
6
```

135

Run

PARTICIPATION ACTIVITY 2.7.2: Modulo.

Determine the result. Type "Error" if appropriate. Only literals appear in these expressions to focus attention on the operators; most practical expressions include variables.

1) `50 % 2`

Check Show answer

2) `51 % 2`

Check Show answer

3) `78 % 10`

Check Show answer

4) `596 % 10`

Check Show answer

5) `100 % (1 // 2)`

Check Show answer

CHALLENGE ACTIVITY 2.7.1: Enter the output of the integer expressions.

566436.4601014.qx3zqy7

Start

Type the program's output

```
x = 14
y = x / 4

print(y)
```

3.5

1 2

Check Next

Modulo examples

Modulo has several useful applications. Below are just a few.

Example 2.7.1: Getting digits.

Given a number, % and // can be used to get each digit. For a three-digit number user_val like 927:


```
ones_digit     = user_val % 10     # Ex: 927 % 10 is 7.
tmp_val        = user_val // 10    #

tens_digit     = tmp_val % 10      # Ex: tmp_val = 927 // 10 is 92. Then 92 %
10 is 2.
tmp_val        = tmp_val // 10

hundreds_digit = tmp_val % 10      # Ex: tmp_val = 92 // 10 = 9. Then 9 % 10
is 9.
```

Example 2.7.2: Get prefix.

Given a 10-digit phone number stored as an integer, % and // can be used to get any part, such as the prefix. For phone_num = 9365551212 (whose prefix is 555):


```
tmp_val = phone_num // 10000  # '// 10000' removes rightmost 4 digits , so
'9365551212 // 10000' is 936555.
prefix_num = tmp_val % 1000 # '% 1000' gets rightmost 3 digits, so '936555 %
1000' is 555.
```

Dividing by a power of 10 removes rightmost digits. Ex: 321 // 10 is 32. Ex: 321 // 100 is 3.

% by a power of 10 gets rightmost digits. Ex: 321 % 10 is 1. Ex: 321 % 100 is 21.

PARTICIPATION ACTIVITY 2.7.3: Modulo examples.

1) Given a non-negative number x, which expression has the range of 5 to 10?

- x % 5
- x % 10
- x % 11
- (x % 6) + 5

2) Given a non-negative number x, which expression has the range -10 to 10?

- x % -10
- (x % 21) - 10
- (x % 20) - 10

3) Which expression gets the tens digit of x? Ex: If x = 693, which expression yields 9?

- x % 10
- x % 100
- (x / / 10) % 10

4) Given a 16-digit credit card number stored in x, which expression gets the rightmost four digits (assume the fourth digit from the right is non-zero)?

- ○ x / 10000
- ○ x % 10000

CHALLENGE ACTIVITY 2.7.2: Division and modulo.

566436.4601014.qx3zqy7

Start

Integers total_sunflowers and num_sunflowers are read from input. A bouquet consists of num_sunflo number of sunflowers is given by total_sunflowers. Assign remaining_sunflowers with the remaining su many bouquets as possible.

▶ **Click here for example**

```
1 total_sunflowers = int(input())
2 num_sunflowers = int(input())
3
4 ''' Your code goes here '''
5
6 print('Remaining sunflowers:', remaining_sunflowers)
```

1 2

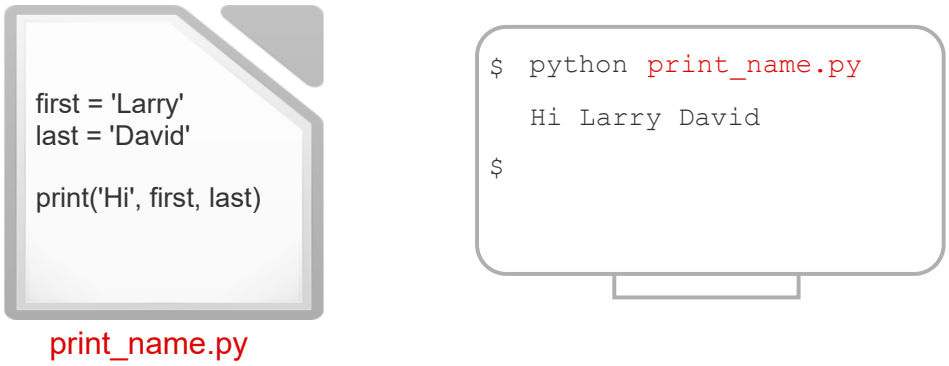
Check Next level

2.8 Module basics

Modules

The interactive Python interpreter allows a programmer to execute one line of code at a time. This method of programming is mostly used for very short programs or for practicing the language syntax. Instead, programmers typically write Python program code in a file called a ***script***, and execute the code by passing the script as input to the Python interpreter.

PARTICIPATION ACTIVITY 2.8.1: Scripts are files executed by the interpreter.



Animation content:

Static figure:
There is a file labeled print_name.py with code and a screen with output.
Begin Python code in file:
first = 'Larry'
last = 'David'

print('Hi', first, last)
End Python code in file.
python print_name.py and Hi Larry David are output on the screen.

Step 1: Programmer writes code in a script file named print_name.py.
The code:
first = 'Larry'
last = 'David'

print('Hi', first, last)
appears in the file labeled print_name.py.

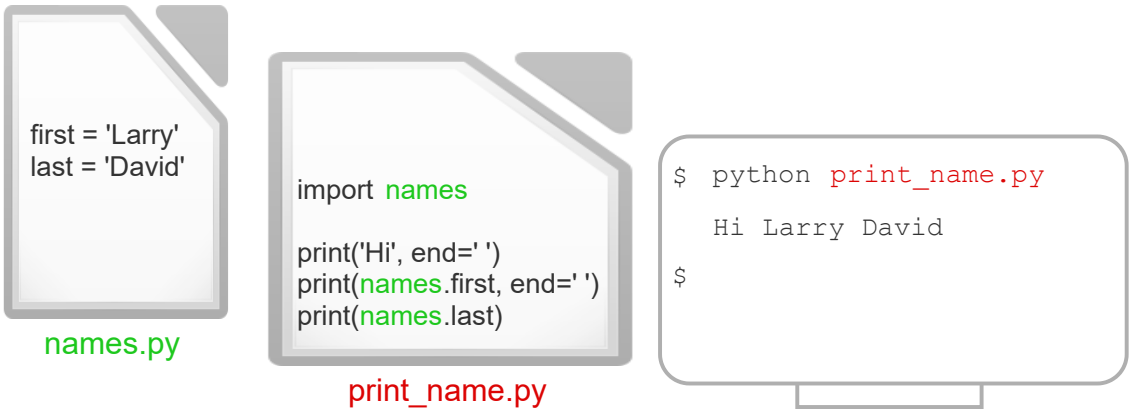
Step 2: The programmer runs the Python interpreter, passing the script as input (shown above using the operating system command line).
python print_name.py is output on the screen.
Hi Larry David is output on the screen.

Animation captions:

1. Programmer writes code in a script file named print_name.py.
2. The programmer runs the Python interpreter, passing the script as input (shown above using the operating system command line).

Programmers often write code in more than just a single script file. Collections of logically related code can be stored in separate files and imported for use into a script that requires that code. A ***module*** is a file containing Python code that can be used by other modules or scripts. A module is made available for use via the ***import*** statement. Once a module is imported, any object defined in that module can be accessed using ***dot notation***. Ex: A variable speed_of_light defined in universe.py is accessed via `universe.speed_of_light`.

PARTICIPATION ACTIVITY 2.8.2: Importing modules.



A script can import and use the contents of a module.

Animation content:

Static figure:
There is a file labeled names.py, a file labeled print_name.py with code and a screen with output.
Begin Python code in file labeled names.py:

first = 'Larry'
last = 'David'
End Python code in file labeled names.py.

Begin Python code in file labeled print_name.py:
import names

print('Hi', end=' ')
print(names.first, end=' ')
print(names.last)
End Python code in file labeled print_name.py.

python print_name.py and Hi Larry David are output on the screen.

Step 1: Code can be separated into multiple files. The names.py module has predefined variables.
File labeled names.py appears with code:
first = 'Larry'
last = 'David'

Step 2: The print_name.py script imports variables from names.py using dot notation.
File labeled print_name.py appears with code:
import names

print('Hi', end=' ')
print(names.first, end=' ')
print(names.last)

Step 3: Running the script imports the module and accesses the module contents using dot notation.
python print_name.py and Hi Larry David are output on the screen.

Animation captions:

1. Code can be separated into multiple files. The names.py module has predefined variables.
2. The print_name.py script imports variables from names.py using dot notation.
3. Running the script imports the module and accesses the module contents using dot notation.

Separating code into different modules makes management of larger programs simpler. For example, a simple Tetris-like game might have a module for input (buttons.py), a module for descriptions of each piece shape (pieces.py), a module for score management (score.py), etc.

The Python standard library, discussed elsewhere, is a collection of useful pre-installed modules. Modules also become more useful when dealing with topics such as functions and classes, where the logical boundaries of what code should be contained within a module is more obvious.

PARTICIPATION ACTIVITY 2.8.3: Basic modules.

If unable to drag and drop, refresh the page.

module **dot notation** **import** **script**

	A file containing Python code that is passed as input to the interpreter.
	A file containing Python code that is imported by a script, module, or interactive interpreter.
	Used to reference an object in an imported module.
	To execute the contents of a file containing Python code and make available the definitions from that file.

Reset

Importing modules and executing scripts

When a module is imported, all code in the module is immediately executed. Python programs often use the built-in special name ***__name__*** to determine if the file was executed as a script by the programmer or imported by another module. If the value of __name__ is the string '__main__', then the file was executed as a script.

In the figure below, two files are provided: pet_names.py initializes some variables, and favorite_pet.py imports pet_names.py as a module and uses some of the variable values to write a message. Running pet_names.py as a script (`python pet_names.py`) causes the code within the `if __name__ == '__main__'` block to execute, which prints some pet statistics. When favorite_pet.py is run and pet_names.py is imported as a module, the pet statistics are not printed.

The *if* construct used in the program below is discussed elsewhere. For now, know that the code indented below the `if __name__ == '__main__'` block only executes when the file is passed to the interpreter directly.

Figure 2.8.1: Checking if a file was executed as a script.

```
# The pet_names.py module

print ('Initializing pet variables...')
pet_name1 = 'Ryder'
pet_name2 = 'Jess'
pet_weight1 = 5.1
pet_weight2 = 8.5

# Executes only if file run as a script (e.g.,
python pet_names.py)
if __name__ == '__main__':
    print('Pet 1:', pet_name1, 'was born',
pet_weight1, 'lbs')
    print('Pet 2:', pet_name2, 'was born',
pet_weight2, 'lbs')
```

```
$ python pet_names.py
Initializing pet
variables...
Pet 1: Ryder was born
5.1 lbs
Pet 2: Jess was born
8.5 lbs
```



```
# A script favorite_pet.py that imports and uses the
pet_names module.

import pet_names  # Importing the module executes
the module contents

print('My favorite pet is', pet_names.pet_name1, '-
')
print('I remember when he weighed only',
pet_names.pet_weight1, 'pounds.')
print('I love', pet_names.pet_name2, 'too, of
course.')
```

```
$ python
favorite_pet.py
Initializing pet
variables...
My favorite pet is
Ryder -
I remember when he
weighed only 5.1
pounds.
I love Jess too, of
course.
```

PARTICIPATION ACTIVITY 2.8.4: Importing modules and executing scripts.

What is the output when running the following commands? Assume valid input of "10" is provided to the program, if required. If no output is generated, select "NO OUTPUT". Note: The math module, imported in fall_time.py, provides functions for advanced math operations and is discussed in more detail elsewhere.

constants.py	fall_time.py

```
# Gravitational constants for
various planets

earth_g = 9.81   # m/s^2
mars_g = 3.71

if __name__ == '__main__':
    print(f'Earth constant:
{earth_g:.2f}')
    print(f'Mars constant:
{mars_g:.2f}')
```

```
# Find seconds to drop from a
height on some planets.
import constants
import math

height = int(input('Height in
meters: '))   # Meters from planet

if __name__ == '__main__':
    print(f'Earth: {math.sqrt(2 *
height / constants.earth_g):.2f}
seconds')
    print(f'Mars: {math.sqrt(2 *
height / constants.mars_g):.2f}
seconds')
```


1) $ python constants.py

- NO OUTPUT
- Earth constant: 9.81
  Mars constant: 3.71
- Height in meters:
  Earth: 1.43 seconds
  Mars: 2.32 seconds

2) $ python fall_time.py

- NO OUTPUT
- Earth constant: 9.81
  Mars constant: 3.71
- Height in meters:
  Earth: 1.43 seconds
  Mars: 2.32 seconds

2.9 Math module

The math module

While basic math operations like + or * are sufficient for some computations, programmers sometimes wish to perform more advanced math operations such as computing a square root. Python comes with a standard ***math module*** to support such advanced math operations. A ***module*** is Python code located in another file. The programmer can import the module for use in their own file, or in an interactive interpreter. The programmer first imports the module to the top of a file.

The math module provides a number of theoretic, trigonometric, and logarithmic operations that a programmer may use. A mathematical operation provided by the math module can be used as follows:

Figure 2.9.1: Importing the math module and calling a math module function.

```
import math

num = 49
num_sqrt =
math.sqrt(num)
```


sqrt() is known as a function. A ***function*** is a list of statements that can be executed simply by referring to the function's name. The statements for sqrt() are within the math module itself and are not relevant to the programmer. The programmer provides a value to the function (like num above). The function executes its statements and returns the computed value. Thus, sqrt(num) above will evaluate to 7.0.

The process of invoking a function is referred to as a ***function call***. The item passed to a function is referred to as an ***argument***. Some functions have multiple arguments, such as the function pow(b, e), which returns b^e. The statement `ten_generation_ancestors = 1024 * num_people` could be replaced by `ten_generation_ancestors = math.pow(2, 10) * num_people` to be more clear.

zyDE 2.9.1: Example of using a math function: Savings interest program.

Note: Blank print statements are used to go to the next line after reading pre-entered input.

Load default template...

```
1  import math
2
3  base = float(input('Enter in
4  print()
5
6  rate = float(input('Enter an
7  print()
8
9  years = int(input('Enter yea
10 print()
11
12 total = base * math.pow(1 +
13
14 print (f'Savings after {year
15
```

```
5000
3.5
20
```

Run

Commonly used functions

Commonly used functions from the math module are listed below. http://docs.python.org/3/library/math.html has a complete listing.

Table 2.9.1: Functions in the standard math module.

Function	Description	Function	Description
Number representation and theoretic functions			
ceil(x)	Round-up value	fabs(x)	Absolute value
factorial(x)	factorial (3! = 3 * 2 * 1)	floor(x)	Round-down value
fmod(x, y)	Remainder of division	fsum(x)	Floating-point sum of a range, list, or array.
Power, exponential, and logarithmic functions			
exp(x)	Exponential function e^x	log(x, (base))	Natural logarithm; base is optional
pow(x, y)	Raise x to power y	sqrt(x)	Square root
Trigonometric functions			

acos(x)	Arc cosine		asin(x)	Arc sine
atan(x)	Arc tangent		atan2(y, x)	Arc tangent with two parameters
cos(x)	Cosine		sin(x)	Sine
hypot(x1, x2, x3, ..., xn)	Length of vector from origin		degrees(x)	Convert from radians to degrees
radians(x)	Convert degrees to radians		tan(x)	Tangent
cosh(x)	Hyperbolic cosine		sinh(x)	Hyperbolic sine
Complex number functions				
gamma(x)	Gamma function		erf(x)	Error function
Mathematical constants				
pi (constant)	Mathematical constant 3.141592...		e (constant)	Mathematical constant 2.718281...

PARTICIPATION ACTIVITY 2.9.1: Variable assignments with math functions.

Determine the final value of z.

1)
```
x = 2.3
z = math.ceil(x)
```

Check Show answer

2)
```
x = 2.3
z = math.floor(x)
```

Check Show answer

3)
```
z = 4.5
z = math.pow(math.floor(z),
2.0)
```

Check Show answer

4)
```
z = 15.75
z = math.sqrt(math.ceil(z))
```

Check Show answer

5)
```
z = 4
z = math.factorial(z)
```

Check Show answer

CHALLENGE ACTIVITY 2.9.1: Math functions.

566436.4601014.qx3zqy7

Start

Type the program's output

```
import math

x = math.sqrt(25.0)
print(x)
```

5.0

1 2 3 4

Check Next

CHALLENGE ACTIVITY 2.9.2: Using math functions to calculate the distance between two points.

Assign point_dist with the distance between point (x1, y1) and point (x2, y2). The calculation is:

$$Distance = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Sample output with inputs: 1.0 2.0 1.0 5.0

```
Points distance: 3.0
```

Note: For nested function calls, ensure each function has an opening and closing parenthesis. Otherwise, a SyntaxError will be raised pointing to the next line of code.

See How to Use zyBooks for info on how our automated program grader works.

566436.4601014.qx3zqy7

```
import math

x1 = float(input())
y1 = float(input())
x2 = float(input())
y2 = float(input())

''' Your solution goes here '''

print(f'Points distance: {point_dist:.1f}')
```

Run

View your last submission

CHALLENGE ACTIVITY 2.9.3: Writing math calculations.

566436.4601014.qx3zqy7

Start

Compute: $answer = p/|q|$

▶ **Click here for example**

Note: Functions in the math module can be accessed using dot notation. Ex: The constant pi is access

```
import math

p = float(input())
q = float(input())

''' Your code goes here '''
```

```
7
8 print(f'answer = {answer:.1f}')  # Outputs answer with 1 decimal place
```


1 2 3

Check Next level

2.10 Random numbers

Generating a random number

Some programs need to use a random number. Ex: A game program may need to roll dice, or a website program may generate a random initial password.

The ***random*** module, in the Python Standard Library, provides methods that return random values. The ***random()*** method returns a random floating-point value each time the function is called, in the range 0 (inclusive) to 1 (exclusive).

The statement `import random` enables use of the random module.

Figure 2.10.1: Outputting two random floating-point numbers.

```
# imports random module
import random

# Generates a random floating point number
print(random.random())
print(random.random())
```

```
0.04675854711720506
0.944076660354665
```

PARTICIPATION ACTIVITY 2.10.1: Random number basics.

1) What import statement is used to access the random number module?
   - ○ import random_number
   - ○ import random
2) The random number returned by `random.random()` will be in what range?
   - ○ 0 (inclusive) to 9 (exclusive)
   - ○ -100 (inclusive) to 100 (exclusive)
   - ○ 0 (inclusive) to 1 (exclusive)

Restricting random integers to a specific number of values using randrange()

Usually, a programmer wants a random integer restricted to a specific number of possible values. Python's ***randrange()*** method generates random integers within a specified range. A single positive integer argument can be passed to the randrange() method to return an integer between 0 (inclusive) and the specified value (exclusive). Ex: `random.randrange(10)` returns an integer with 10 possible values: 0, 1, 2, ..., 8, 9.

PARTICIPATION ACTIVITY 2.10.2: Restricting random integers to a specific number of values.

```
import random

# Generates random integers with 3 possible values
print(random.randrange(3))
print(random.randrange(3))
print(random.randrange(3))
print(random.randrange(3))
print(random.randrange(3))
```

```
2
1
0
0
2
```


Animation content:

The code shows:

```
import random
print ("Random numbers")

# Generates random integers with 3 possible values
print(random.randrange(3))
print(random.randrange(3))
print(random.randrange(3))
print(random.randrange(3))
print(random.randrange(3))
```


The output is:

```
2
1
0
0
2
```

Animation captions:

1. Each random.randrange(3) statement returns a random integer with 3 possible values: 0, 1, and 2 .

PARTICIPATION ACTIVITY 2.10.3: randrange() method.

1) random.randrange(20) returns an integer with how many possible values?
   - 21
   - 20
   - 19

2) What is the smallest *possible* value returned by random.randrange(10)?
   - 0
   - 1
   - 9

3) What is the largest *possible* value returned by random.randrange(45)?
   - 44

- 45
- 46

4) Which expression generates a random integer in the range 0 to 30?

- random.randrange(29)
- random.randrange(30)
- random.randrange(31)

5) Which expression best mimics the random outcome of flipping a coin?

- random.randrange(1)
- random.randrange(2)
- random.randrange(3)

Defined ranges

The technique above generates random integers with N possible values ranging from 0 to N-1. Ex: 6 values from 0 to 5. A programmer usually requires a range starting with a non-zero value x. Ex: 10 to 15, or -20 to 20. Two methods in the random module, randint() and randrange(), can produce random integers within a defined range.

Table 2.10.1: Random number modules for a defined range.

Method	Code
randint(min, max) returns a random integer between min and max inclusive.	`# Returns a random integer between 12 and 20 inclusive` `print(random.randint(12, 20))`
randrange(min, max) returns a random integer between min and max - 1 inclusive.	`# Returns a random integer between 12 and 19 inclusive` `print(random.randrange(12, 20))`

randint() an alias for randrange()

randint() is actually an alias for randrange(), meaning that the randint(min, max) is defined in the random module as randrange(min, max+1).

PARTICIPATION ACTIVITY 2.10.4: Generating random integers in a defined range not starting from 0.

Animation content:

A programmer attempts to use two methods to generate a random integer between 2 to 10 inclusive. The first expression shown, random.randrange(2,10), generates random values from 2 to 9. While the second expression, random.randint(2,10), generates random values in the specified 2 to 10 range.

Animation captions:

1. A programmer wants to generate a random integer in the range 2 to 10 inclusive.
2. random.randrange(2,10) is only capable of generating 8 values in the range from 2 to 9.
3. The expression random.randint(2,10) is capable of generating all 9 values in the range from 2 to 10.

PARTICIPATION ACTIVITY 2.10.5: Generating random integers in a specific range.

1) The expression random.randrange(_____) returns one of 6 possible integers between 0 and 5.

Check Show answer

2) The expression random.randrange(_____) produces a random integer between 0 and 4.

Check Show answer

3) How many possible values can be produced by random.randint(10, 15)?

Check Show answer

4) How many possible values can be produced by random.randint(-5, 5)?

Check Show answer

5) The expression random.randint(_____) produces 9 possible values between 4 and 12.

Check Show answer

6) How many values are in the range -4 to 0 using the expression random.randrange(-4, 0) ?

Check Show answer

Example: Randomly moving a student

The following program randomly moves a student from one seat to another seat in a lecture hall, perhaps to randomly move students before an exam. The seats are in 20 rows numbered 1 to 20. Each

row has 30 seats (columns) numbered 1 to 30. The student should be moved from the left side (columns 1 to 15) to the right side (columns 16 to 30).

Figure 2.10.2: Randomly moving a student from one seat to another.

```
#imports random module
import random
"""
Switch a student from a random seat on the left
(cols 1 to 15)
to a random seat on the right (cols 16 to 30)
Seat rows are 1 to 20
"""
rowNumL = random.randint(1, 20)  # 1 to 20 rows left
colNumL = random.randint(1, 15)  # 1 to 15  columns
left
rowNumR = random.randint(1, 20)  # 1 to 20 rows right
colNumR = random.randint(16, 30) # 16 to 30 columns
right

print(f'Move from row {rowNumL} col {colNumL} to row
{rowNumR} col {colNumR}')
```

```
Move from row 18 col 13
to row 13 col 16
```


PARTICIPATION ACTIVITY 2.10.6: Random integer example: Moving seats.

Consider the above example.

1) The expression `random.randint(1, 20)` is used to generate a random row integer. random.randint(1, 20) is _____.
   - inclusive
   - exclusive

2) The expression `random.randint(1, _____ )` generates a random column within the left half.
   - 15
   - 20

3) The random right column can be generated with random.randrange(16, 30).
   - True

False

Pseudo-random

The numbers generated by the random module are known as pseudo-random. "Pseudo" means "not actually, but having the appearance of." Internally, the random module has an equation to compute the next "random" number from the previous one, (invisibly) keeping track of the previous one. For the first call to any random method, no previous random number exists, so the method uses a built-in integer based on the current time, called a ***seed***, to help generate a random number. Since the time is different for each program run, each program will get a unique sequence.

Reproducibility is important for testing some programs. Ex: Despite the appearance of randomness, classic arcade games like Pac-Man allow players to master the game by repeating winning behaviors. A programmer can specify the seed by calling the seed() method. Ex: random.seed(5). With a specific seed, each program run yields the same sequence of pseudo-random numbers.

Figure 2.10.3: Using the same seed for each program run.

```
#import random method
import random

# Generates a unique seed
random.seed(15)

#Generates a random integer between 1, 10
print (random.randint(1, 10))
print (random.randint(1, 10))
print (random.randint(1, 10))
```

```
4
1
9

(next run)

4
1
9
```

PARTICIPATION ACTIVITY 2.10.7: Using a unique seed for each program run.

1) In the absence of a seed being supplied initially, the expression `random.randint(x, y)` yields a different integer sequence with each successive iteration.

   - True
   - False

2) Generating a random integer with the expression `random.randint(0, 5)`

using `random.seed(3)` yields a different integer sequence for each program run.

- True
- False

3) randint() generates a "pseudo-random" sequence of values because the sequence begins repeating itself after about 20 numbers.

- True
- False

2.11 Representing text

Unicode

String variables represent text, such as the character "G" or the word "pineapple". Python uses ***Unicode*** to represent every possible character as a unique number, known as a ***code point***. For example, the character "G" has the code point decimal value of 71. Below is a table with Unicode code points and the character represented by each code point. In total, there are over 1 million code points in the Unicode standard character set.

Table 2.11.1: Encoded text values.

Decimal	Character	Decimal	Character	Decimal	Character
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f

39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_

103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

©zyBooks 11/21/24 13:02 2300507
Liz Vokac Main
KVCCCIS216JohnsonFall2024

PARTICIPATION ACTIVITY 2.11.1: Unicode.

1) What is the decimal encoding of the "{" character?

Check Show answer

Escape sequences

In addition to visible characters such as "a", "$", or "5", several special characters exist. A ***newline*** character, which indicates the end of a line of text, is encoded as 10. Since there is no visible character for a newline, the language uses the two-item sequence \n to represent a newline character. The \ is known as a ***backslash***. Upon reaching a \, the interpreter recognizes that item as the start of a special character's two-item sequence and then looks at the next item to determine the special character. The two-item sequence is called an ***escape sequence***.

Table 2.11.2: Common escape sequences.

Escape Sequence	Explanation	Example code	Output
\\	Backslash (\)	`print('\\home\\users\\')`	`\home\users\`
\'	Single quote (')	`print('Name: John O\'Donald')`	`Name: John O'Donald`
\"	Double quote (")	`print("He said, \"Hello friend!\"")`	`He said, "Hello friend!"`
\n	Newline	`print('My name...\nIs John...')`	`My name...` `Is John...`
\t	Tab (indent)	`print('1. Bake cookies\n\t1.1. Preheat oven')`	`1. Bake cookies` `    1.1. Preheat oven`

PARTICIPATION ACTIVITY 2.11.2: Escape sequences.

1) What is the output of `print('\\c\\users\\juan')`

- \\c\\users\\juan
- \c\users\juan
- \\\\c\\users\\\\juan

2) What is the output of `print('My name is \'Tater Tot\'.')`

- My name is Tater Tot.
- My name is "Tater Tot".
- My name is 'Tater Tot'.

3) What is the output of `print('10...\n9...')`

- 10...9...
- 10...
  9...
- 10...\n9...

Raw strings and converting between an encoding and text

Escape sequences can be ignored using a ***raw string***. A raw string is created by adding an "r" before a string literal, as in `r'this is a raw string\''`, which would output as `this is a raw string\'`.

Figure 2.11.1: Ignoring escape characters with a raw string.

```
my_string = 'This is a \n \'normal\' string\n'
my_raw_string = r'This is a \n \'raw\' string'

print(my_string)
print(my_raw_string)
```

```
This is a
 'normal' string

This is a \n \'raw\' string
```

Sometimes converting between a text character and the encoded value is useful. The built-in function ***ord()*** returns an encoded integer value for a string of length one. The built-in function ***chr()*** returns a string of one character for an encoded integer.

PARTICIPATION ACTIVITY 2.11.3: Using ord() to convert a character to the encoded value.

Type any character and observe the output of the ord() function, which is the numerical encoding of the character. Try upper- and lowercase letters, as well as special characters like "%" or "$", or a space (should result in "32"). Try copy/pasting any one of these characters from the Korean Unicode character set: 강 남 스 타 일.

Type a character: ord(' A ') Encoded number: **65**

PARTICIPATION ACTIVITY 2.11.4: Using chr() to convert an encoded value to a character.

Type any number greater than or equal to 0 and observe the encoded value's character equivalent. Note that not all numbers will result in a visible character.

Type a number (0-255): `chr(` 90 `)`

ASCII char: Z

CHALLENGE ACTIVITY 2.11.1: Enter the output of the print() statements.

566436.4601014.qx3zqy7

Start

Type the program's output

```
print('The name of the book is "1984."')
```

```
The name of the book is
```


1 2 3 4

Check Next

PARTICIPATION ACTIVITY 2.11.5: Text.

1) Complete the code to output

\\

```
print(                )
```

Check Show answer

2) Use a raw string literal to assign "C:\file.doc" to my_str (without quotes).

```
my_str =
```

Check Show answer

CHALLENGE ACTIVITY 2.11.2: Representing text.

566436.4601014.qx3zqy7

Start

String location_input is read from input. String my_str contains a special character. Modify the string lit adding a backslash (\) to the special character so the program outputs `Xim isn't going to the f`

Ex: If the input is `store`, then the output is:

```
Xim isn't going to the store
```

```
1 location_input = input()
2
3 # Modify the string literal below
4 my_str = 'Xim isn't going to the'
5
6 print(my_str, location_input)   # Output my_str followed by location_input, separated by
```

1 2

Check Next level

Exploring further:

- Unicode HOWTO from the official Python documentation.

2.12 Additional practice: Number games

The following are sample programming lab activities. Not all classes using a zyBook require students to complete these activities. No auto-checking is performed. Users planning to complete these programs may consider first developing their code in a separate programming environment.

Repeating digits game

Several math games manipulate numbers in simple ways that yield fun results. Below is a program that takes any given two-digit number and outputs a six-digit number, repeating the two digits. Ex: 24 becomes 242424.

zyDE 2.12.1: Number game: Repeating digits.

Enter a two-digit number into the input box and press the run button.

Load default template...

```
1 num = int(input('Enter 2 digi
2
3 result = num * (3 * 7 * 13 *
4
5 print(result)
6
```

24

Run

Create a different version of the program that:

1. Takes a three-digit number and generates a six-digit number with the three-digit number repeated. Ex: 391 becomes 391391. The rule is to multiply the three-digit number by 7*11*13.
2. Takes a five-digit number and generates a ten-digit number with the five-digit number repeated. Ex: 49522 becomes 4952249522. The rule is to multiply the five-digit number by 11*9091.

Multiplying by 11 game

A two-digit number can be easily multiplied by 11 simply by adding the digits and inserting the sum between the digits. Ex: 43 * 11 has the resulting digits of 4, 4+3, and 3, yielding 473. If the sum between the digits is greater than 9, then the 1 is carried to the hundreds place. Complete the below program.

zyDE 2.12.2: Number game: Multiplying by 11.

Load default template...

```
# Complete the following program

num_in_tens = int(input('Enter the tens digit:\n'))
num_in_ones = int(input('Enter the ones digit:\n'))

num_in = num_in_tens * 10 + num_in_ones

print('You entered', num_in)
print(num_in, '* 11 is', num_in*11)

num_out_hundreds = num_in_tens + ((num_in_tens + num_in_or
#num_out_tens = ?   FINISH
#num_out_ones = ?  FINISH

print('An easy mental way to find the answer is:')
print(num_in_tens, ',', num_in_tens, '+', num_in_ones, ','
#Build num_out from its digits:
#num_out = ? + ? + ?   FINISH
```



```
5
8
```

Run

2.13 LAB: Divide input integers

Write a program that reads integers `user_num` and `div_num` as input, and outputs `user_num` divided by `div_num` three times using floor divisions.

Ex: If the input is:

```
2000
2
```


the output is:

```
1000 500 250
```

Note: In Python 3, floor division discards fractions. Ex: 6 // 4 is 1 (the 0.5 is discarded).

LAB ACTIVITY | 2.13.1: LAB: Divide input integers | 10 / 10

main.py — Load default template...

```
1  user_num = int(input())
2  div_num = int(input())
3
4  quotient1 = user_num // div_num
5  quotient2 = quotient1 // div_num
6  quotient3 = quotient2 // div_num
7
8  print(quotient1, quotient2, quotient3)
9
10
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

```
9/17 T10 min:3
```

2.14 LAB: Driving costs

Driving is expensive. Write a program with a car's gas mileage (miles/gallon) and the cost of gas (dollars/gallon) as floating-point input, and output the gas cost for 20 miles, 75 miles, and 500 miles.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

```
print(f'{your_value1:.2f} {your_value2:.2f} {your_value3:.2f}')
```


Ex: If the input is:

```
25.0
3.1599
```

where the gas mileage is 25.0 miles/gallon and the cost of gas is $3.1599/gallon, the output is:

```
2.53 9.48 63.20
```

Note: Real per-mile cost would also include maintenance and depreciation.

LAB ACTIVITY | 2.14.1: LAB: Driving costs | 10 / 10

main.py — Load default template...

```
gas_mileage = float(input())
gas_price_per_gl = float(input())

gas_cost = gas_price_per_gl / gas_mileage

gas_cost20 = gas_cost * 20
gas_cost75 = gas_cost * 75
gas_cost500 = gas_cost * 500


print(f'{gas_cost20:.2f} {gas_cost75:.2f} {gas_cost500:.2f}')
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Program output displayed here

Coding trail of your work What is this?

```
9/17 T10 min:7
```

2.15 LAB: Expression for calories burned during workout

The following equation estimates the average calories burned for a person when exercising, which is based on a scientific journal article (source):

$$Calories = \frac{(Age \times 0.2757 + Weight \times 0.03295 + HeartRate \times 1.0781 - 75.4991) \times Ti}{8.368}$$

Write a program using inputs age (years), weight (pounds), heart rate (beats per minute), and time (minutes), respectively. Output the average calories burned for a person.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

```
print(f'Calories: {calories:.2f} calories')
```

Ex: If the input is:


```
49
155
148
60
```

then the output is:

```
Calories: 736.21 calories
```

566436.4601014.qx3zqy7

LAB ACTIVITY | 2.15.1: LAB: Expression for calories burned during workout | 10 / 10

main.py

Load default template...

```
1  ''' Calories = ((Age x 0.2757) + (Weight x 0.03295) + (Heart Rate x 1.0781) – 75.4991)
2
3  patient_age = int(input())
4  weight = float(input())
5  heart_rate = int(input())
6  time = float(input())
7
8  calories = (((patient_age * 0.2757) + (weight * 0.03295) + (heart_rate * 1.0781) - 75.
9
10 print(f'Calories: {calories:.2f} calories')
11
12
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → Output

Program output displayed here

Coding trail of your work [What is this?](#)

```
9/17 T10 min:4
```

2.16 LAB: Using math functions

Given three floating-point numbers x, y, and z, output x^z, x^{y^z}, the absolute value of (x minus y), and the square root of x^z.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

```
print(f'{your_value1:.2f} {your_value2:.2f} {your_value3:.2f} {your_value4:.2f}')
```

Ex: If the input is:

```
5.0
1.5
3.2
```

Then the output is:

```
172.47 361.66 3.50 13.13
```


LAB ACTIVITY | 2.16.1: LAB: Using math functions | 10 / 10

main.py — Load default template...

```
1  import math
2  ''' X^z  XtoYtoZ,  |x-y| sqrt(x^z)'''
3
4  x = float(input())
5  y = float(input())
6  z = float(input())
7
8  x_pow_z = math.pow(x, z)
9  x_pow_y_pow_z = math.pow(x, math.pow(y, z))
10 abs_val = math.fabs(x - y)
11 sqrt = math.sqrt(math.pow(x, z))
12
13 print(f'{x_pow_z:.2f} {x_pow_y_pow_z:.2f} {abs_val:.2f} {sqrt:.2f}')
14
15
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

```
9/17 T10 min:9
```

2.17 LAB: Musical note frequencies

On a piano, a key has a frequency, say f0. Each higher key (black or white) has a frequency of f0 * r^n, where n is the distance (number of keys) from that key, and r is $2^{(1/12)}$. Given an initial key frequency, output that frequency and the next 3 higher key frequencies.

Output each floating-point value with two digits after the decimal point, then the units ("Hz"), then a newline, using the following statement:
`print(f'{your_value:.2f} Hz')`

Ex: If the input is:

```
440
```


(which is the A key near the middle of a piano keyboard), the output is:

```
440.00 Hz
466.16 Hz
493.88 Hz
523.25 Hz
```

Note: To compute the next 3 higher key frequencies, use one statement to compute r = $2^{(1/12)}$ using the pow function (remember to import the math module). Then use that r in subsequent statements that use the formula fn = f0 * r^n with n being 1, 2, and finally 3.

566436.4601014.qx3zqy7

LAB ACTIVITY 2.17.1: LAB: Musical note frequencies 0 / 10

main.py Load default template...

```
1 ''' Type your code here. '''
2
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → Output

Program output displayed here

Coding trail of your work What is this?