1.1 Basic graphics

Creating a graphics frame

Python supports a set of objects for developing graphical applications. A ***graphical application*** is a program that displays drawings and other graphical objects. TKinter is a standard Python package for graphical applications. TKinter displays contents inside a window called a ***frame*** using a ***Frame*** object. The following program shows how to create and configure a Frame object to display an empty application window.

Figure 1.1.1: Creating a Frame object for a graphical application.

```
import tkinter as tk

class Application(tk.Frame):
   def __init__(self, master=None):
       super().__init__(master)
       self.master = master

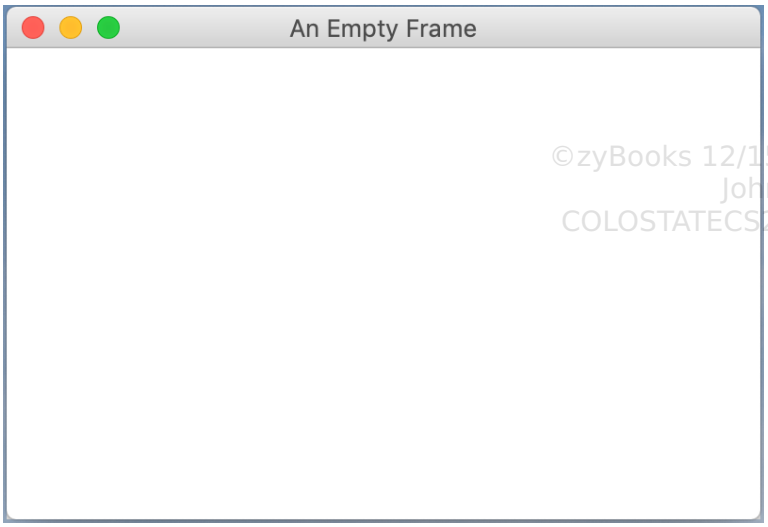
       # Set the frame's title
       self.master.title('An Empty Frame')
       self.pack()

app_frame = tk.Tk()

# Set the frame's width (400) and height (250) in
pixels
app_frame.geometry('400x250')

# Make the frame visible to the user
app = Application(master=app_frame)
app.mainloop()
```

Figure 1.1.2: Screenshot of empty application window.



Constructing a Frame object does not immediately display a frame. The program uses the methods supported by the Frame object to configure and display the frame as follows:

1. **Set the frame's size** by calling the geometry() method with arguments for the width and height, as in `app_frame.geometry('400x250')`. Forgetting to set the frame's size results in a frame too small to see.
2. **Set the frame's title** by calling the title() method with a String as the argument.
3. **Make the frame visible** to the user by calling the mainloop() method.

PARTICIPATION ACTIVITY 1.1.1: Configuring a Frame.

Select the code statement that would resolve the described problem. Assume an empty Frame object named appFrame.

1) The frame window lacks a title. User would like the title to be "My program".

   - `self.master.title(My program)`
   - `self.master.title('My program')`

2) The program called the mainloop() method correctly, but the frame is not visible on the screen. The frame should be 500 pixels wide and 300 pixels tall.

- `app_frame.geometry('500x300')`
- `app.mainloop(false);`
- `app_frame.geometry('300x500')`

Drawing graphical objects

A Frame can be used to draw graphical objects, such as rectangles, circles, and lines. To display graphical objects, a programmer can add a Canvas object to a frame. A ***Canvas*** is a graphical component that a programmer can use to draw basic shapes.

The following program demonstrates how to build a class that creates a Canvas to draw 2D graphics.

Figure 1.1.3: Basic example showing how to create a class to draw 2D graphics using Canvas.

```
import tkinter as tk
from tkinter import Canvas, Frame,
BOTH

class Application(tk.Frame):
   def __init__(self, master=None):
       super().__init__(master)
       self.master = master
       self.pack(fill=BOTH, expand=1)

       canvas = Canvas(self)
       # Write your drawing
instructions

app_frame = tk.Tk()
app_frame.geometry('400x250')
app = Application(master=app_frame)
app.mainloop()
```


The above code defines a class named Application that uses a Canvas object. A programmer completes the template by providing custom drawing instructions after the Canvas object has been created. In the animation below, the programmer uses Canvas's create_rectangle() to draw a rectangle in the frame.

Canvas's create_rectangle() takes the following arguments:

1. **Arguments 1 and 2:** coordinate for the top left corner (x0, y0) of the rectangle
2. **Arguments 3 and 4:** coordinate for the bottom right corner (x1, y1)

3. **Argument 5 (optional)**: rectangle outline color (if not set, the outline will be black)
4. **Argument 6 (optional):** rectangle fill color

Many more optional arguments exist for the create_rectangle() method, such as width (width of border) and dash (make the border dashed).

PARTICIPATION ACTIVITY 1.1.2: Drawing a filled rectangle.

Animation content:

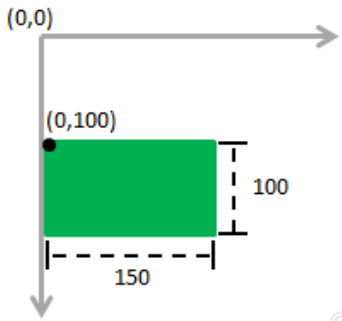
undefined

Animation captions:

1. The canvas variable is set to a Canvas object for drawing 2D graphics.
2. The create_rectangle() method draws a rectangle with the coordinates (10, 75) and (150, 50), with the outline and fill color set to "cyan".

The programmer needs to know the positioning coordinate system in order to draw shapes in the intended location. As the following figure illustrates, the top-left corner of a Canvas corresponds to coordinates (0, 0). The x-coordinate increases horizontally to the right and the y-coordinate increases vertically downward.

Figure 1.1.4: Graphics coordinate system.



PARTICIPATION ACTIVITY 1.1.3: Drawing colored rectangles.

Select the code statement that would resolve the described problem. Assume canvas is a Canvas object.

1) The user wants a rectangle's top left corner coordinates to be x0=5 and y0=5 and the rectangle to be 100 x 100 pixels in size.

   - `canvas.create_rectangle(5, 5, 105, 105)`
   - `canvas.create_rectangle(5, 5, 100, 100)`
   - `canvas.create_rectangle(0, 0, 100, 100)`

2) The user wants a pink rectangle to have a blue outline.

   - `canvas.create_rectangle(10, 10, 200, 50, outline='pink', fill='blue')`
   - `canvas.create_rectangle(10, 10, 200, 50, outline='blue', fill='pink')`
   - `canvas.create_rectangle(10, 10, 200, 50, outline='blue', color='pink')`

Ex: Drawing a basic histogram

The following program uses a Canvas object to draw a simple histogram using Canvas's create_rectangle() method. The program first creates a HistogramViewer object named histogram_viewer and adds the object to the Frame. The HistrogramViewer class creates three rectangles by calling Canvas's create_rectangle() method.

Figure 1.1.5: Drawing a histogram in a frame.

HistogramApp.py

```
import tkinter as tk
from tkinter import Canvas, Frame, BOTH

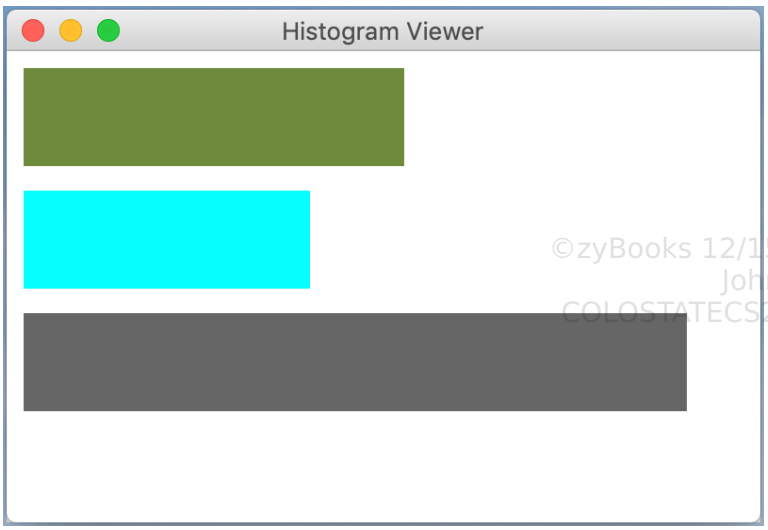
class HistogramViewer(tk.Frame):
   def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.master.title('Histogram Viewer')
        self.pack(fill=BOTH, expand=1)

        canvas = Canvas(self)
        canvas.create_rectangle(10, 10, 210, 60,
outline='darkolivegreen4', fill='darkolivegreen4')
        canvas.create_rectangle(10, 75, 160, 125, outline='cyan',
fill='cyan')
        canvas.create_rectangle(10, 140, 360, 190, outline='gray40',
fill='gray40')

        canvas.pack(fill=BOTH, expand=1)
        self.pack()

app_frame = tk.Tk()
app_frame.geometry('400x250')
histogram_viewer = HistogramViewer(master=app_frame)
histogram_viewer.mainloop()
```


Figure 1.1.6: Screenshot of HistogramViewer application.



PARTICIPATION ACTIVITY 1.1.4: Drawing rectangles.

Which code segment (type the number) performs the described operation? Assume the Canvas object is called graphicsObj.

1. `graphicsObj.create_rectangle(0, 0, 150, 100, fill='green')`
2. `graphicsObj.create_rectangle(0, 100, 200, 300, outline='red', fill='red')`
3. `graphicsObj.create_rectangle(0, 100, 50, 250, outline='purple', fill='purple')`

1) Draws a filled in square.

Check Show answer

2) Draws a rectangle 50 pixels wide and 150 pixels in height.

Check Show answer

3) Draws a rectangle whose top-left corner is located at the origin of the coordinate system.

Check Show answer

4) Draws a rectangle with a black outline.

Check Show answer

©zyBooks 12/15/22 00:10 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Canvas provides methods for drawing structured graphics, of which some common shapes are summarized below:

Table 1.1.1: Summary of common shapes for drawing.

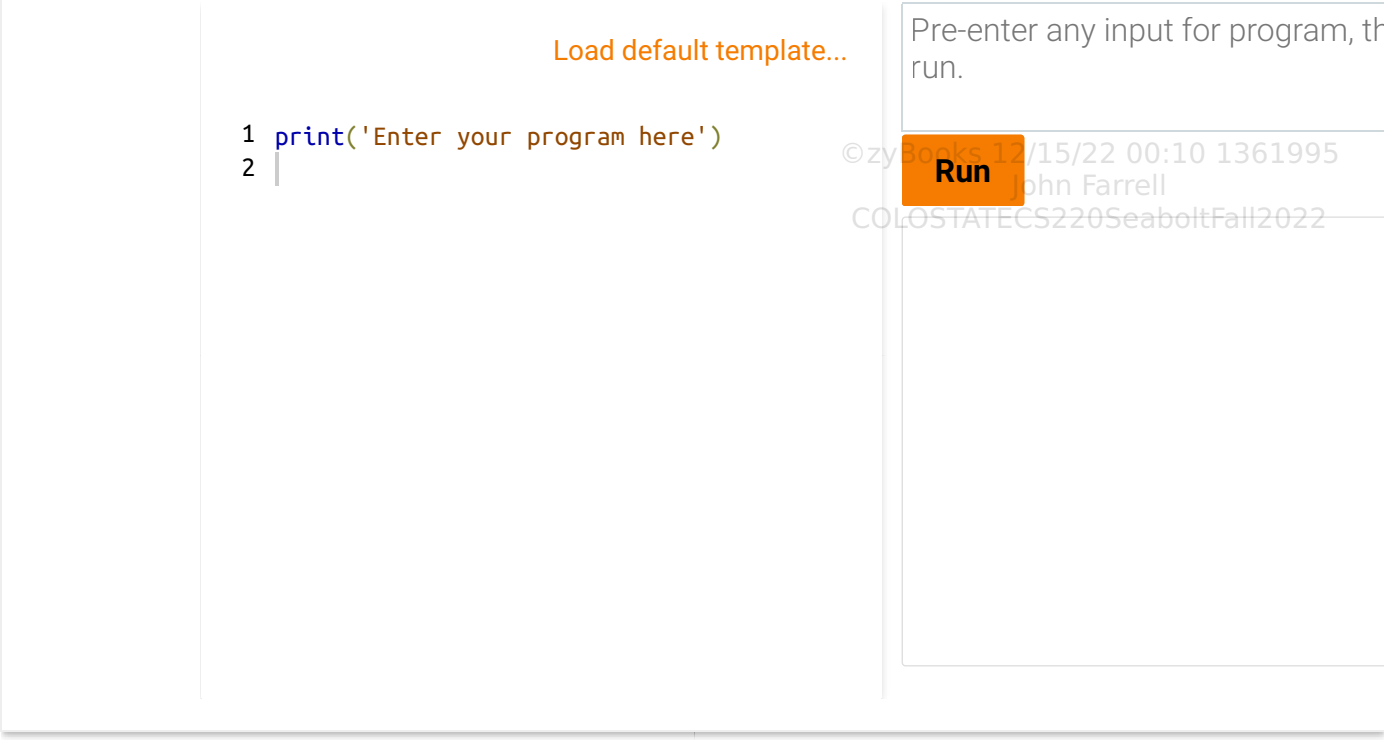
Shape	Description	Documentation
Rectangle	Draws a rectangle on the canvas.	create_rectangle() method
Oval	Draws and ellipse on the canvas.	create_oval() method
Line	Draws a line on the canvas.	create_line() method
Polygon	Draws a polygon on the canvas.	create_polygon() method

Exploring further:

- Color chart for TKinter (all available color names)

1.2 zyBooks built-in programming window

zyDE 1.2.1: Programming window.



1.3 Basic input and output

This section has been set as optional by your instructor.

Basic text output

Printing output to a screen is a common programming task. This section describes basic output; later sections have more details.

The primary way to print output is to use the built-in function ***print()***. Ex: `print('hello world')`. Text enclosed in quotes is known as a ***string literal***. Text in string literals may have letters, numbers, spaces, or symbols like @ or #. Each use of `print()` starts on a new line.

A string literal can be surrounded by matching single or double quotes: `'Python rocks!'` or `"Python rocks!"`. Good practice is to use single quotes for shorter strings and double quotes for more complicated text or text that contains single quotes, like `print("Don't eat that!")`.

Figure 1.3.1: Printing text.

```
# Each print statement starts on a new line
print('Hello there.')
print('My name is...')
print('Carl?')
```

```
Hello there.
My name
is...
Carl?
```


PARTICIPATION ACTIVITY 1.3.1: Basic text output.

1) Select the statement that prints the following: *Welcome!*

- print(Welcome!)
- print('Welcome!")
- print('Welcome!')

PARTICIPATION ACTIVITY 1.3.2: Basic text output.

1) Type a statement that prints the following: *Hello*

Check Show answer

CHALLENGE ACTIVITY 1.3.1: Output simple text.

Write the simplest statement that prints the following:

```
3 2 1 Go!
```

Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

422102.2723990.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```


Run

View your last submission

CHALLENGE ACTIVITY 1.3.2: Output an eight with asterisks.

Complete the program with four more print statements to output the following figure with asterisks. Do not add spaces after the last character on each line.

```
*****
*   *
*****
*   *
*****
```

Note: Whitespace (blank spaces/blank lines) matters; make sure your whitespace matches *exactly* the expected output.

422102.2723990.qx3zqy7

```
1 print('*****')
2
3 ''' Your solution goes here '''
4
```

Run

View your last submission

Keeping output on the same line

Each call to `print()` outputs on a new line. However, sometimes a programmer may want to keep output on the same line. A programmer can add `end=' '` inside of `print()` to keep the output of a subsequent print statement on the same line separated by a single space. Ex: `print('Hello', end=' ')`.

Figure 1.3.2: Printing text on the same row.

```
# Including end=' ' keeps output on same
line
print('Hello there.', end=' ')
print('My name is...', end=' ')
print('Carl?')
```

```
Hello there. My name is...
Carl?
```

PARTICIPATION ACTIVITY 1.3.3: Printing text on the same row.

1) Which pair of statements print output on the same line?

- print('Halt!')
  print('No access!')
- print('Halt!', end=' ')
  print('No access!')
- print(Halt!, end=' ')
  print(No Access!, end=' ')

Outputting a variable's value

The value of a variable can be printed out via: `print(variable_name)` (without quotes around variable_name).

Figure 1.3.3: Printing the value of a variable.


```
wage = 20

print('Wage is', end=' ')
print(wage)  # print variable's value
print('Goodbye.')
```

```
Wage is
20
Goodbye.
```

PARTICIPATION ACTIVITY 1.3.4: Basic variable output.

1) Given the variable num_cars = 9, which statement prints 9?

- print(num_cars)
- print("num_cars")

PARTICIPATION ACTIVITY 1.3.5: Basic variable output.

1) Write a statement that prints the value of the variable num_people.

Check Show answer

Outputting multiple items with one statement

Programmers commonly try to use a single print statement for each line of output by combining the printing of text, variable values, and new lines. A programmer can simply separate the items

with commas, and each item in the output will be separated by a space. Combining string literals, variables, and new lines can improve program readability, because the program's code corresponds more closely to the program's printed output.

Figure 1.3.4: Printing multiple items using a single print statement.


```
wage = 20

print('Wage:', wage)  # Comma separates multiple
items
print('Goodbye.')
```

```
Wage: 20
Goodbye.
```

A common error is to forget the comma between items, as in `print('Name' user_name)`.

Newline characters

Output can be moved to the next line by printing `"\n"`, known as a ***newline character***. Ex: `print('1\n2\n3')` prints "1" on the first line, "2" on the second line, and "3" on the third line of output. `"\n"` consists of two characters, \ and n, but together are considered by the Python interpreter as a single character.

Figure 1.3.5: Printing using newline characters.

```
print('1\n2
\n3')
```

```
1
2
3
```

`print()` always adds a newline character after the output automatically to move the next output to the next row, unless `end=' '` is provided to replace the newline character with a space (or some other character). An empty `print()` can be used to print only a newline.

Figure 1.3.6: printing without text.

```
print('123')
print()
print('abc')
```

```
123

abc
```


Any space, tab, or newline is called ***whitespace***.

NOTE: In a normal programming environment, program input is provided interactively and completed by pressing the enter key. The enter key press would insert a newline. Since zyBooks input is pre-entered, no enter key press can be inferred. Thus, activities that require pre-entered input may need extra newline characters or blank print statements in zyBooks, compared to other environments.

PARTICIPATION ACTIVITY 1.3.6: Output simulator.

The tool below supports a subset of Python, allowing for experimenting with print statements. The activity is marked as complete upon interacting with the tool.

The variables `country_population = 1344130000` and `country_name = 'China'` have been defined and can be used in the simulator.

Try printing the following output:

```
The population of China was 1344130000 in 2011.
```

Remember, commas can be used to output multiple items with a single print statement.
Ex:
`print('The person lives in', country_name, 'with their family.')`
outputs `The person lives in China with their family.`

```
print('Change this string!')
```



```
Change this string!
```

PARTICIPATION ACTIVITY 1.3.7: Single print statement.

Assume variable age = 22, pet = "dog", and pet_name = "Gerald".

1) What is the output of
```
print('You are', age,
'years old.')
```

Check Show answer

2) What is the output of
```
print(pet_name, 'the', pet,
'is', age)
```

Check Show answer

CHALLENGE ACTIVITY 1.3.3: Enter the output.

Type the program's output. Remember that print() outputs an ending newline, so press Enter or Return on your keyboard to indicate a newline in your answer.

422102.2723990.qx3zqy7

Start

Type the program's output

```
print('Bob is happy.')
```

1 2 3

Check Next

Basic input

Many useful programs allow a user to enter values, such as typing a number, a name, etc.

Reading input is achieved using the ***input()*** function. The statement `best_friend = input()`

will read text entered by the user and the best_friend variable is assigned with the entered text. The function input() causes the interpreter to wait until the user has entered some text and has pushed the return key.

The input obtained by input() is any text that a user typed, including numbers, letters, or special characters like # or @. Such text in a computer program is called a ***string***.

A string simply represents a sequence of characters. For example, the string 'Hello' consists of the characters 'H', 'e', 'l', 'l', and 'o'. Similarly, the string '123' consists of the characters '1', '2', and '3'.

PARTICIPATION ACTIVITY 1.3.8: A program can get an input value from the keyboard.

Animation captions:

1. The input() statement gets an input value from the keyboard and puts that value into the best_friend variable.
2. best_friend's value can then be used in subsequent processing and outputs.

PARTICIPATION ACTIVITY 1.3.9: Reading user input.

1) Which statement reads a user-entered string into variable num_cars?

- `num_cars = input`
- `input() = num_cars`
- `num_cars = input()`

PARTICIPATION ACTIVITY 1.3.10: Reading user input.

1) Complete a statement that reads a user-entered string into variable my_var.

Check Show answer

Converting input types

The string '123' (with quotes) is fundamentally different from the integer 123 (without quotes). The '123' string is a sequence of the characters '1', '2', and '3' arranged in a certain order, whereas 123 represents the integer value one-hundred twenty-three. Strings and integers are each an example of a ***type***; a type determines how a value can behave. For example, integers can be divided by 2, but not strings (what sense would `"Hello" / 2` make?). Types are discussed in detail later on.

Reading from input always results in a string type. However, often a programmer wants to read in an integer, and then use that number in a calculation. If a string contains only numbers, like '123', then the ***int()*** function can be used to convert that string to the integer 123.

Figure 1.3.7: Using int() to convert strings to integers.

```
my_string = '123'
my_int =
int('123')

print(my_string)
print(my_int)
```

```
123
123
```

A programmer can combine input() and int() to read in a string from the user and then convert that string to an integer for use in a calculation.

Figure 1.3.8: Converting user input to integers.

```
print('Enter wage:', end='
')
wage = int(input())

new_wage = wage + 10
print('New wage:',
new_wage)
```

```
Enter wage:
8
New wage: 18
```


PARTICIPATION ACTIVITY 1.3.11: Converting user input to integers.

1) Type a statement that converts the string '15' to an integer and assigns my_var with the result.

Check Show answer

2) Complete the code so that new_var is equal to the entered number plus 5.

```
my_var = int(input())
new_var =
```

Check Show answer

Input prompt

Adding a string inside the parentheses of input() displays a prompt to the user before waiting for input and is a useful shortcut to adding an additional print statement line.

Figure 1.3.9: Basic input example.

```
hours = 40
weeks = 52
hourly_wage = int(input('Enter hourly wage:
'))

print('Salary is', hourly_wage * hours *
weeks)
```

```
Enter hourly wage:
12
Salary is 24960
...
Enter hourly wage:
20
Salary is 41600
```

NOTE: The below tool requires input to be pre-entered. This is a current limitation of the web-based tool and atypical of conventional Python environments, where users enter input as the program runs. For conventional behavior, you may copy-paste the program into a local environment, such as IDLE.

zyDE 1.3.1: Basic input.

Run the program and observe the output. Change the input box value from 3 to anoth number, and run again.

Load default ter


```
human_years = int(input('Enter age of dog (in human years): '))
print()

dog_years = 7 * human_years

print(human_years, 'human years is about', end=' ')
print(dog_years, 'dog years.')


```

3

Run

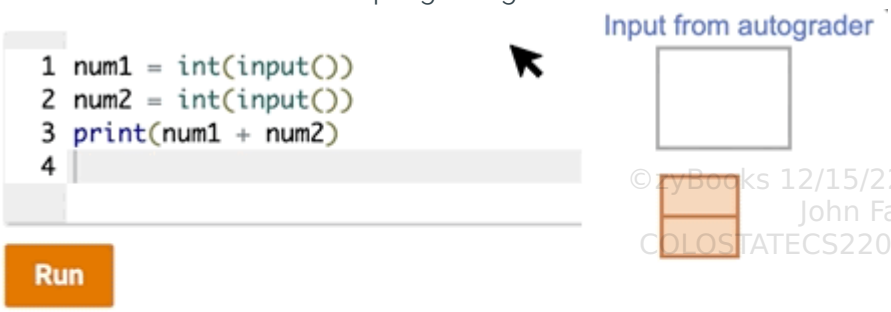
CHALLENGE ACTIVITY 1.3.4: Read user input numbers and perform a calculation.

The following program reads in 2 numbers from input, assigns them to num1 and num2 respectively, and then outputs the sum of those numbers. Copy the code provided to see how this code is executed in an autograded system.


```
num1 = int(input())
num2 = int(input())
print(num1 + num2)
```

Note: Our autograder automatically runs your program several times, trying different input

values each time to ensure your program works for any values. This program is tested twice, first with the inputs 5 and 10, and then with the inputs 6 and 3. See How to Use zyBooks for info on how our automated program grader works.



422102.2723990.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```

Run

CHALLENGE ACTIVITY 1.3.5: Read user input and print to output.

Read three integers from user input **without a prompt**. Then, print the product of those integers. Ex: If input is 2 3 5, output is 30.

Note: Our system will run your program several times, automatically providing different input values each time, to ensure your program works for any input values. See How to Use zyBooks for info on how our automated program grader works.

422102.2723990.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```


Run

CHALLENGE ACTIVITY 1.3.6: Output basics.

For activities with output like below, your output's whitespace (newlines or spaces) must match exactly. See this note.

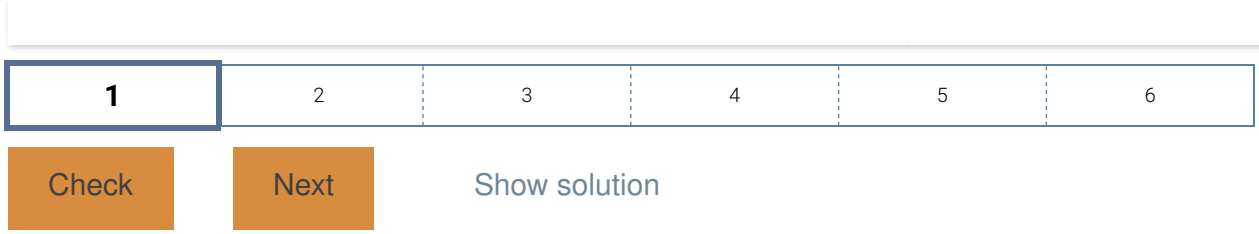
422102.2723990.qx3zqy7

Start

Write code that outputs the following. End with a newline. Remember that print() automatically adds a newline.

```
This week was wonderful.
```

```
1
2 ''' Your code goes here '''
3
```



Check Next Show solution

1.4 Arithmetic expressions

This section has been set as optional by your instructor.

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 1.4.1: 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 1.4.1: 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 1.4.2: 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 such order is known in programming as ***precedence rules***, listed below.

Table 1.4.2: 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: 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 1.4.3: Evaluating expressions.

Animation captions:

1. An expression like 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 such order is not obvious.

PARTICIPATION ACTIVITY 1.4.4: Evaluating expressions and precedence rules.

Select the expression whose 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

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 1.4.5: 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 1.4.1: Calculate the values of the integer expressions.

422102.2723990.qx3zqy7

Start

Type the program's output

```
x = 17
y = x + 2

print(y)
```

1 2 3 4 5

Check Next

1.5 Detecting equal values with branches

This section has been set as optional by your instructor.

Detecting if two items are equal using an if statement

A program commonly needs to determine if two items are equal. Ex: If a hotel gives a discount for guests on their 50th wedding anniversary, a program to calculate the discount can check if a variable numYears is equal to the value 50. A programmer can use an if statement to check if two values are equal.

An ***if*** statement executes a group of statements if an expression is true. The statements in a branch must be indented some number of spaces, typically four spaces.

The example below uses ==. The ***equality operator*** (***==***) evaluates to true if the left and right sides are equal. Ex: If numYears is 50, then numYears == 50 evaluates to true. Note the equality operator is ==, not =.

PARTICIPATION ACTIVITY 1.5.1: Detecting if two items are equal: Hotel discount.

Animation content:

undefined

Animation captions:

1. An if statement executes a group of statements if an expression is true. The program assigns hotel_rate with 150 and then gets the number of years the user has been married from input.
2. num_years is 50. So the expression num_years == 50 evaluates to true, and the if's statement will execute. The statements after the colon : will execute next.
3. hotel_rate is divided in half, which is the discount for guests celebrating their 50th wedding anniversary.
4. The program completes by printing the hotel rate.

PARTICIPATION ACTIVITY 1.5.2: If statement.

What is the final value of num_items?

1)
```
bonus_val = 10
num_items = 1

if bonus_val == 10:
    num_items = num_items
+ 3
```


Check Show answer

2)
```
bonus_val = 0
num_items = 1

if bonus_val == 10:
    num_items = num_items
+ 3
```

Check Show answer

Equality and inequality operators

Whereas the equality operator checks whether two values are equal, the ***inequality operator*** (***!=***) evaluates to true if the left and right sides are not equal, or different.

An expression involving an equality or inequality operator evaluates to a Boolean value. A ***Boolean*** is a type that has just two values: True or False.

Table 1.5.1: Equality and inequality operators.

Equality operators	Description	Example (assume x is 3)
==	a ***==*** b means a is equal to b	x == 3 is True x == 4 is False
!=	a ***!=*** b means a is not equal to b	x != 3 is False x != 4 is True

PARTICIPATION ACTIVITY 1.5.3: Evaluating expressions that have equality operators.

Indicate whether the expression evaluates to True or False.
x is 5, y is 7.

1) x == 5

- ○ True
- ○ False

2) x == y

- True
- False

3) y != 7

- True
- False

4) y != 99

- True
- False

5) x != y

- True
- False

PARTICIPATION ACTIVITY 1.5.4: Creating expressions with equality operators.

Type the equality or inequality operator to make the expression true.

1) num_dogs is 0.

`num_dogs` [ ] `0`

Check **Show answer**

2) num_dogs and num_cats are the same.

`num_dogs` [ ] `num_cats`

Check **Show answer**

3) num_dogs and num_cats differ

`num_dogs` [ ] `num_cats`

Check **Show answer**

4) num_dogs is either less than or

greater than num_cats.

```
num_dogs [ ] num_cats
```

Check Show answer

5) user_char is the character 'x'.

```
user_char [ ] 'x'
```

Check Show answer

If-else statement

An ***if-else*** statement executes one group of statements when an expression is true, and another group of statements when the expression is false. In the example below, the if-else statement outputs if a number entered by the user is even or odd. The if statement executes if divRemainder is equal to 0, and the else statement executes if divRemainder is not equal to 0.

PARTICIPATION ACTIVITY 1.5.5: If-else statement: Determining if a number is even or odd.

Animation content:

The program shown:

```
user_num = int(input('Enter a number: '))

div_remainder = user_num % 2

if div_remainder == 0:
    print(user_num, 'is even.')
else:
    print(user_num, 'is odd.')
```

Console input/output shown from 2 runs of the program:

```
Enter a number: 22
22 is even.

Enter a number: 45
45 is odd.
```

Animation captions:

1. An if-else statement executes a group of statements if an expression is True, and executes another group of statements otherwise.
2. user_num % 2 evaluates to the remainder of dividing user_num by 2. user_num is 22, so div_remainder is assigned with 0.
3. The if statement's expression div_remainder == 0 evaluates to 0 == 0, which is True. So the if statements execute.
4. user_num is 45, so div_remainder is assigned with 1. The if statement's expression div_remainder == 0 evaluates to 1 == 0, which is False. So the else's statements execute.

PARTICIPATION ACTIVITY 1.5.6: If-else statements.

1) What is the final value of num_items?

```
bonus_val = 12

if bonus_val == 12:
    num_items = 100
else:
    num_items = 200
```

Check Show answer

2) What is the final value of num_items?

```
bonus_val = 11

if bonus_val == 12:
    num_items = 100
else:
    num_items = 200
```

Check Show answer

3) What is the final value of num_items?

```
bonus_val = 15
num_items = 44

if bonus_val == 14:
    num_items = num_items
+ 3
else:
    num_items = num_items
= num_items + 1
```

Check Show answer

4) What is the final value of bonus_val?

```
bonus_val = 11

if bonus_val != 12:
   bonus_val = bonus_val
+ 1
else:
   bonus_val = bonus_val
+ 10
```

Check Show answer

5) What is the final value of bonus_val?

```
bonus_val = 12

if bonus_val == 12:
    bonus_val = bonus_val
+ 2
    bonus_val = 3 *
bonus_val

else:
    bonus_val = bonus_val
+ 10
```

Check Show answer

PARTICIPATION ACTIVITY 1.5.7: Writing an if-else statement.

Translate each description to an if-else statement as directly as possible. (Not checked, but please indent a branch's statements some consistent number of spaces, such as 3 spaces.)

1) If user_age equals 62, assign item_discount with 15. Else, assign item_discount with 0.

Check Show answer

2) If num_people equals 10, execute group_size = 2 * group_size. Otherwise, execute group_size = 3 * group_size and num_people = num_people - 1.

Check Show answer

3) If num_players does not equal 11, execute team_size = 11. Otherwise, execute team_size = num_players. Then, no matter the value of num_players, execute team_size = 2 * team_size.

CHALLENGE ACTIVITY 1.5.1: Branches with equality and inequality operators: Enter the output.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.5.2: Basic if-else.

422102.2723990.qx3zqy7

Multi-branch if-else statements

Commonly, a program may need to detect several specific values of a variable. An if-else statement can be extended to have three (or more) branches. Each branch's expression is checked in sequence. As soon as one branch's expression is found to be true, that branch's statement executes (and no subsequent branch is considered). If no expression is true, the else branch executes. The example below detects values of 1, 25, or 50 for variable num_years.

Figure 1.5.1: Multi-branch if-else statement. Only 1 branch will execute.

```
if expression1:
   # Statements that execute when expression1 is true
   # (first branch)
elif expression2:
   # Statements that execute when expression1 is false and expression2 is
true
   # (second branch)
else:
   # Statements that execute when expression1 is false and expression2 is
false
   # (third branch)
```

Figure 1.5.2: Multi-branch if-else example: Anniversaries.

```
num_years = int(input('Enter number years
married: '))

if num_years == 1:
    print('Your first year -- great!')
elif num_years == 10:
    print('A whole decade -- impressive.')
elif num_years == 25:
    print('Your silver anniversary -- enjoy.')
elif num_years == 50:
    print('Your golden anniversary --
amazing.')
else:
    print('Nothing special.')
```

```
Enter number years married:
10
A whole decade --
impressive.

...

Enter number years married:
25
Your silver anniversary --
enjoy.

...

Enter number years married:
30
Nothing special.

...

Enter number years married:
1
Your first year -- great!
```


PARTICIPATION ACTIVITY 1.5.8: Multi-branch if-else statements.

What is the final value of employee_bonus for each given value of num_sales?

```
if num_sales == 0:
    employee_bonus = 0
elif num_sales == 1:
    employee_bonus = 2
elif num_sales == 2:
    employee_bonus = 5
else:
    employee_bonus = 10
```

1) num_sales is 2

Check Show answer

2) num_sales is 0

Check Show answer

3) num_sales is 7

Check Show answer

1.6 Detecting ranges with branches (general)

Detecting ranges using if-elseif-else

A common programming task is to detect if a value lies within a certain range and then perform an action depending on where the value lies. Ex: If Timmy is less than 6, he can play pee-wee soccer. If Timmy is between 6 and 17, he can play junior league soccer, and if he's older than 17, he can play professional soccer.

An if-elseif-else structure can detect number ranges with each branch performing a different action for each range. Each expression only needs to indicate the upper range part; if execution reaches an expression, the lower range part is implicit from the previous expressions being false.

PARTICIPATION ACTIVITY 1.6.1: An if-elseif-else structure can elegantly detect ranges.

Animation captions:

1. Kids of various ages may wish to play soccer. A soccer club may not have teams for kids 5 and under.
2. One level of teams is listed as "Under 8" (or just U8), which is understood to mean just 7 or 6, but not 5 or younger.
3. Likewise, U10 means 9 and 8, and U12 means 11 and 10. No teams exist for ages 12 and over.
4. An if-elseif-else structure can elegantly capture such ranges. When an expression is checked, the reviewer knows that all previous expressions are false, thus defining the low-range end.

PARTICIPATION ACTIVITY 1.6.2: Using if-elseif-else to detect increasing ranges.

Indicate the range corresponding to each branch. x is a non-negative integer.

If unable to drag and drop, refresh the page.

10 - 19 **30+** **20 - 29** **0 - 9**

If x < 10 : Branch 1

Else If x < 20 : Branch 2

Else If x < 30 : Branch 3

Else : Branch 4

Reset

PARTICIPATION ACTIVITY 1.6.3: More ranges with if-elseif-else.

Indicate the range detected by the expression, assuming each question continues a single if-elseif-else structure. Type ranges as: 25 - 29

1) If x > 100 : Branch 1

`- infinity`

Check **Show answer**

2) Else If x > 50 : Branch 2

Check **Show answer**

3) Else

`-infinity -`

Check **Show answer**

4) Is this a reasonable if-elseif-else structure? Type yes or no.

If x < 100: Branch 1
Else If x < 200: Branch 2
Else If x < 150: Branch 3
Else: Branch 4

Check Show answer

CHALLENGE ACTIVITY 1.6.1: Decision sequence to detect increasing ranges.

422102.2723990.qx3zqy7

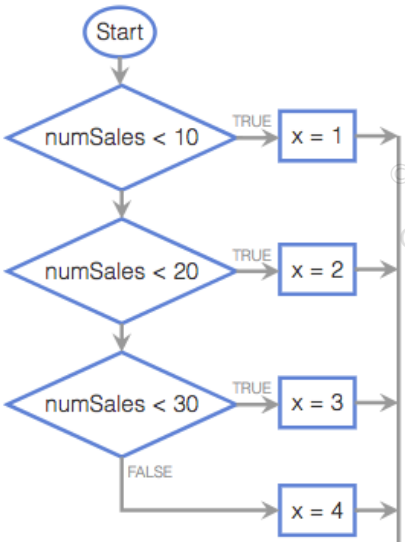
Using multi-branch if-else to detect ranges

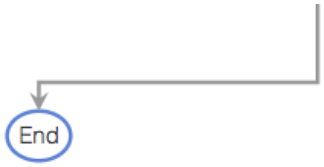
The sequential nature of multi-branch if-else statements is useful to detect ranges of numbers. In the following example, the second branch expression is only reached if the first expression is false. So the second branch is taken if userAge < 16 *is false* (so 16 or greater) AND userAge is < 25, meaning userAge is between 16 - 24 (inclusive).

PARTICIPATION ACTIVITY 1.6.4: Using if-elseif for ranges: Insurance prices.

PARTICIPATION ACTIVITY 1.6.5: Decision sequences and ranges.

Type the range for each branch. Type ranges as 25 - 29, or as 30+ for 30 and up.





1) Range for x = 2

Check Show answer

2) Range for x = 3

Check Show answer

3) Range for x = 4

Check Show answer

CHALLENGE ACTIVITY 1.6.2: Flowchart decision sequence to detect increasing ranges.

422102.2723990.qx3zqy7

1.7 Detecting ranges with branches

Relational operators

A ***relational operator*** checks how one operand's value relates to another, like being greater than.

Some operators, like >=, involve two characters. A programmer cannot arbitrarily combine the >, =, and < symbols; only the shown two-character sequences represent valid operators.

Table 1.7.1: Relational operators.

Relational operators	Description	Example (assume x is 3)
<	a **<** b means a is less than b	x < 4 is True x < 3 is False
>	a **>** b means a is greater than b	x > 2 is True x > 3 is False
<=	a **<=** b means a is less than or equal to b	x <= 4 is True x <= 3 is True x <= 2 is False
>=	a **>=** b means a is greater than or equal to b	x >= 2 is True x >= 3 is True x >= 4 is False

PARTICIPATION ACTIVITY 1.7.1: Evaluating equations having relational operators.

Indicate whether the expression evaluates to True or False.
x is 5, y is 7.

1) x <= 7

- ○ True
- ○ False

2) y >= 7

- ○ True
- ○ False

3) Is x <> y a valid expression?

- ○ Yes
- ○ No

4) Is x =< y a valid expression?

- Yes
- No

PARTICIPATION ACTIVITY 1.7.2: Creating expressions with relational operators.

Type the operator to complete the desired expression.

1) num_dogs is greater than 10

`num_dogs` [ ] `10`

Check Show answer

2) num_cars is greater than or equal to 5

`num_cars` [ ] `5`

Check Show answer

3) num_cars is 5 or greater

`num_cars` [ ] `5`

Check Show answer

4) cents_lost is a negative number

`cents_lost` [ ] `0`

Check Show answer

Detecting ranges with if-else statements

Programmers commonly use the sequential nature of the multi-branch if-else arrangement to detect ranges of numbers. In the following example, the second branch expression is only reached if the first expression is false. So the second branch is taken if user_age < 16 *is False* (so 16 or greater) AND user_age is < 25, meaning user_age is between 16 - 24 (inclusive).

PARTICIPATION ACTIVITY 1.7.3: Using the sequential nature of multi-branch if-else for ranges: Insurance prices.

Animation content:

undefined

Animation captions:

1. The user enters 27 for their age, which is stored in memory as the variable user_age. The multi-branch if-else first checks if user_age is less than 16, which is False.
2. The next branch in the multi-branch if-else checks if user_age is less than 25, which is False.
3. The next branch checks if user_age is less than 40, which is True. The elif's statements execute and the variable insurance_price is set to 2350 in memory.

PARTICIPATION ACTIVITY 1.7.4: Ranges and multi-branch if-else.

Type the range for each branch. Type ranges as: 25 - 29, or type 30+ for all numbers 30 and larger.

```
if num_sales < 10:
    ...
elif num_sales < 20:   # 2nd branch range: _____
    ...
elif num_sales < 30:   # 3rd branch range: _____
    ...
else:                  # 4th branch range: _____
    ...
```

1) 2nd branch range:

Check Show answer

2) 3rd branch range:

Check Show answer

3) 4th branch range:

Check Show answer

4) What is the range for the last branch below?

```
if num_items < 0:
    ...
elif num_items > 100:
    ...
else:   # Range: _____
    ...
```


Check Show answer

PARTICIPATION ACTIVITY 1.7.5: Complete the multi-branch if-else.

1) Second branch: user_num is less than 200

```
if user_num < 100 :
    ...
elif  :
    ...
else :  # user_num >= 200
    ...
```

Check Show answer

2) Second branch: user_num is positive. (non-zero)

```
if user_num < 0 :

    ...

                          :
    ...
else :  # user_num is 0

if user_num < 100 :

    ...

                          :

    ...

else :  # user_num is
between

      # 100 and 105

    ...
```

3) The second branch: user_num is greater than 105.

Check Show answer

Check Show answer

4) If the final else branch executes, what must user_num have been? Type "unknown" if appropriate.

```
if user_num <= 9:
    ...
elif user_num >= 11:
    ..
else:
    ... # user_num if this
executes?
```

Check Show answer

5) Which branch will execute? Valid answers: 1, 2, 3, or none.

```
user_num = 555;

if user_num < 0:
    ... # Branch 1
    num < 100:
    # Branch 3
```

Check Show answer

CHALLENGE ACTIVITY 1.7.1: Enter the output for the branches with relational operators.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.7.2: Basic if-else expression.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.7.3: Relational expressions.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.7.4: Detect ranges using branches.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.7.5: Multi-branch if-else statements: Print century.

Write an if-else statement with multiple branches.
If year is 2101 or later, print "Distant future" (without quotes). Otherwise, if year is 2001 or greater, print "21st century". Otherwise, if year is 1901 or greater, print "20th century". Else (1900 or earlier), print "Long ago".

Sample output with input: 1776

```
Long ago
```

422102.2723990.qx3zqy7

```
1 year = int(input())
2
3 ''' Your solution goes here '''
4
```


Run

View your last submission

Operator chaining

Python supports ***operator chaining***. For example, `a < b < c` determines whether b is greater-than a but less-than c. Chaining performs comparisons left to right, evaluating a < b first. If the result is True, then b < c is evaluated next. If the result of the first comparison a < b is False, then there is no need to continue evaluating the rest of the expression. Note that a is not compared to c.

PARTICIPATION ACTIVITY 1.7.6: Chaining relational operators.

Write a relational expression using operator chaining.

1) x is less than y but greater than z

Check Show answer

2) x is a non-negative number less than 100.

```
if :
    # evaluated to True
else:
    # evaluated to False
```

CHALLENGE ACTIVITY 1.7.6: If-else expression: Operator chaining.

Write an expression that will print "in high school" if the value of user_grade is between 9 and 12 (inclusive).

Sample output with input: 10

```
in high school
```

422102.2723990.qx3zqy7

```
1 user_grade = int(input())
2 if ''' Your solution goes here ''':
3     print('in high school')
4 else:
5     print('not in high school')
```

Run

View your last submission

1.8 Detecting ranges using logical operators

Logical AND, OR, and NOT (general)

A ***logical operator*** treats operands as being True or False, and evaluates to True or False. Logical operators include AND, OR, and NOT. Programming languages typically use various symbols for those operators, but below the words AND, OR, and NOT are used for introductory purposes.

PARTICIPATION ACTIVITY 1.8.1: Logical operators: AND, OR, and NOT.

Animation captions:

1. AND evaluates to True only if BOTH operands are True.
2. OR evaluates to True if ANY operand is True (one, the other, or both).
3. NOT evaluates to the opposite of the operand.
4. Each operand is commonly an expression itself. If x = 7, y = 9, then (x > 0) AND (y < 10) is True and True, so evaluates to True (both operands are True).

Table 1.8.1: Logical operators.

Logical operator	Description
a AND b	***Logical AND***: True when both of its operands are True.
a OR b	***Logical OR***: True when at least one of its two operands are True.
NOT a	***Logical NOT***: True when its one operand is False, and vice-versa.

PARTICIPATION ACTIVITY 1.8.2: Evaluating expressions with logical operators.

Indicate whether the expression evaluates to True or False.
x is 7, y is 9.

1) x > 5

- ○ True
- ○ False

2) (x > 5) AND (y < 20)

- True
- False

3) (x > 10) AND (y < 20)

- True
- False

4) (x > 10) OR (y < 20)

- True
- False

5) (x > 10) OR (y > 20)

- True
- False

6) NOT (x > 10)

- True
- False

7) NOT ((x > 5) AND (y < 20))

- True
- False

Detecting ranges with logical operators (general)

A common use of logical operators is to detect if a value is within a range.

PARTICIPATION ACTIVITY 1.8.3: Using AND to detect if a value is within a range.

Animation captions:

1. The range 10 < x < 15 means that x may be 11, 12, 13, 14.
2. Specifying that range in a program can be done using two < operators along with an AND operator. 10 < x defines the range 11 and higher.
3. x < 15 defines the range 14 and lower. ANDing yields the overlapping range. Only when x is 11, 12, 13, or 14 will both expressions be true.

PARTICIPATION ACTIVITY 1.8.4: Using AND to detect if a value is within a range.

Assume x is an integer.

1) Which approach uses a logical operator to detect if x is in the range 1 to 99?
   - 0 < x < 100
   - (0 < x) AND (x < 100)
   - (0 < x) AND (x > 100)

2) Which detects if x is in the range -4 to +4?
   - (x < -5) AND (x < 5)
   - (x > -5) OR (x < 5)
   - (x > -5) AND (x < 5)

3) Which detects if x is either less than -5, or greater than 10?
   - (x < -5) AND (x > 10)
   - (x < -5) OR (x > 10)

Booleans and logical operators

A ***Boolean*** refers to a value that is either True or False. Note that True and False are keywords in Python and must be capitalized. A programmer can assign a Boolean value by specifying True or False, or by evaluating an expression that yields a Boolean.

Figure 1.8.1: Creating a Boolean.

```
my_bool = True   # Assigns my_bool with the boolean value True

is_small = my_val < 3  # Assigns is_small with the result of the
expression (False)
```


Keywords ***and***, ***or***, and ***not*** (lowercase) are used to represent the AND, OR, and NOT logical operators. Logical operators are commonly used in expressions of if-else statements.

Table 1.8.2: Logical operators.

Logical operator	Description
a **and** b	***Boolean AND***: True when both operands are True.
a **or** b	***Boolean OR***: True when at least one operand is True.
not a	***Boolean NOT*** (opposite): True when the single operand is False (and False when operand is True).

Table 1.8.3: Logical operators examples.

Given age = 19, days = 7, user_char = 'q'

`(age > 16) and (age < 25)`	True, because both operands are True.
`(age > 16) and (days > 10)`	False, because both operands are not True (days > 10 is False).
`(age > 16) or (days > 10)`	True, because at least one operand is True (age > 16 is True).
`not (days > 10)`	True, because operand is False.
`not (age > 16)`	False, because operand is True.
`not (user_char == 'q')`	False, because operand is True.

PARTICIPATION ACTIVITY | 1.8.5: Logical operators: Complete the expressions to detect the desired range.

1) days_logged is greater than 30 and less than 90

```
if (days_logged > 30)
    (days_logged <
90):
```

2) 0 < max_cars < 100

Check Show answer

```
if (max_cars > 0)
(max_cars < 100):
```

Check Show answer

3) num_stores is between 10 and 20, inclusive.

```
if (num_stores >= 10)
    (num_stores <=
20):
```

Check Show answer

4) not_valid is either less than 15, or greater than 79.

```
if (not_valid < 15)
    (not_valid > 79):
```

Check Show answer

PARTICIPATION ACTIVITY 1.8.6: Creating expressions with logical operators.

1) num_dogs has a minimum of 2 and a maximum of 5.

```
if (num_dogs >= 2)
                    :
```

Check Show answer

2) wage is greater than 10 and less than 18. Use > and < (not >= and <=). Use parentheses around

sub-expressions.

```
if                    :
```

Check Show answer

3) num is a 3-digit positive integer. Ex: 100, 989, and 523, are 3-digit positive integers, but 55, 1000, and -4 are not.

For most direct readability, your expression should compare directly with the smallest and largest 3-digit number.

```
if (num >= 100)
                    
:
```

Check Show answer

Example: TV channels

A cable TV provider may have regular channels numbered 2-499, and high-definition channels numbered 1002-1499. A program may set a character variable to 's' for standard, 'h' for high-definition, and 'e' for error.

Figure 1.8.2: Detecting ranges: Cable TV channels.

```
if (user_channel >= 2) and (user_channel <= 499):
   channel_type = 's'

elif (user_channel >= 1002) and (user_channel <=
1499):
   channel_type = 'h'

else:
   channel_type = 'e'
```

zyDE 1.8.1: Detecting ranges: Cable TV channels.

Run the program and observe the output. Change the input box value from 3 to anoth number, and run again.

Load default template...

```
1  user_channel = int(input())
2
3  if (user_channel >= 2) and (user_channel <=
4      channel_type = 's'
5
6  elif (user_channel >= 1002) and (user_chan
7      channel_type = 'h'
8
9  else:
10     channel_type = 'e'
11
12 print('Channel type:', channel_type)
13
```

3

Run

PARTICIPATION ACTIVITY 1.8.7: TV channel example: Detecting ranges.

Consider the above example.

1) If user_channel is 300, to what does the if statement's expression, `(user_channel >= 2) and (user_channel <= 499)`, evaluate?

   - true
   - false

2) If user_channel is 300, does the else if's expression `(user_channel >= 1002) and (user_channel <= 1499)` get checked?

   - Yes
   - No

3) Did the expressions use logical AND or logical OR?

- AND
- OR

4) Channels 500-599 are pay channels. Does this expression detect that range? `(user_channel >= 500) or (user_channel <= 599)`

- Yes
- No

Detecting ranges implicitly vs. explicitly

A programmer often uses logical operators to detect a range by explicitly specifying the high-end and low-end of the range. However, if a program should detect increasing ranges without gaps, a multi-branch if-else statement can be used without logical operators; the low-end of the range is implicitly known upon reaching an expression. Likewise, a decreasing range without gaps has implicitly-known high-ends.

PARTICIPATION ACTIVITY 1.8.8: Detecting ranges implicitly vs. explicitly.

Animation content:

`undefined`

Animation captions:

1. This code detects ranges explicitly using the AND operator. The first branch executes when x < 0, the second when (x >=0) and (x <= 10).
2. But, if the first branch doesn't execute, x must be >= 0. So the second branch's expression can just be x <= 10. The x >= 0 is implicit.
3. Implicit ranges can simplify a multi-branch if statement for ranges without gaps.

PARTICIPATION ACTIVITY 1.8.9: Detecting ranges implicitly vs explicitly.

For each pair of statements, does the second if-else statement detect the same ranges as the first if-else statement?

1)

```
if temp <= 0...
elif (temp > 0) and (temp <
100)...
```

```
if temp <= 0...
elif temp < 100...
```

- Yes
- No

2)

```
if systolic < 130: ...
elif (systolic >= 130) and
(systolic <= 139): ...
```

```
if systolic < 130: ...
elif systolic >= 130: ...
```

- Yes
- No

3)

```
if (year >= 1901) and (year <=
2000): ...
elif (year >= 2001) and (year
<= 2100): ...
```

```
if year <= 2000: ...
elif year <= 2100: ...
```

- Yes
- No

CHALLENGE ACTIVITY 1.8.1: Detect number range.

Write an expression that prints "Eligible" if user_age is between 18 and 25 inclusive.
Ex: 17 prints "Ineligible", 18 prints "Eligible".

422102.2723990.qx3zqy7

```
1 user_age = int(input())
2
3 if ''' Your solution goes here ''':
4     print('Eligible')
5 else:
```

Run

View your last submission

1.9 Detecting ranges with gaps

Basic ranges with gaps

Oftentimes, ranges contain gaps. Ex: Movie theaters often give ticket discounts to children (anyone 12 and under) and seniors (anyone 65 and older). The gap is the group of people aged 13 to 64. An if-else statement can be used to detect such ranges with gaps.

PARTICIPATION ACTIVITY | 1.9.1: Using multi-branch if-else for detecting ranges with gaps: Movie ticket prices.

Animation content:

`undefined`

Animation captions:

1. After the user enters their age, the else-if branch's first branch checks if age is <=12.
2. user_age is 67, which is greater than 12, so the program moves to the second branch that checks if user_age is >= 65.
3. 67 is >= 65, so the second branch's statements execute, applying the senior discount to the ticket price. The program concludes by outputting the ticket price.
4. If the user's age falls between the gap of 12 and 65 (13 to 64), the else branch executes and the ticket price is $14, the most expensive price.

PARTICIPATION ACTIVITY 1.9.2: Detecting ranges with gaps and multi-branch if-else.

Select the correct answers below.

1) In the animation above, what is the age range for a child ticket discount?

- 0 - 12
- less than 13
- less than 11

2) In the animation above, what is the age range for a senior ticket discount?

- 65 or more
- 66 or more
- 13 - 64

3) What is the range for the last branch below?

```
if num_items <= 0:
    ...
elif num_items > 100:
    ...
else:  # Range: _____
    ...
```

- 1 - 99
- 0 - 100
- 1 - 100

4) What is the range for the last branch below?

```
if num_items < 50:
    ...
elif num_items > 50:
    ...
else:  # Range: _____
    ...
```

- 49 - 51
- 0 - 50
- 50

Ranges with gaps using logical operators

Programmers often use logical operators to explicitly detect ranges with an upper and lower bound, including ranges with gaps that may have intermediate bounds. Ex: If a valid office number is within the ranges of 100 to 150 or 200 to 250, the logical AND operator or operator chaining can be used to identify the lower and upper bounds of the two ranges. Further, the ranges can be combined into a single branch using the logical OR operator.

PARTICIPATION ACTIVITY 1.9.3: Explicit ranges with gaps detection using logical AND and OR.

Animation content:

undefined

Animation captions:

1. The logical AND operator is used to identify the lower and upper bounds of the two valid ranges of office numbers (100 to 150 and 200 to 250). Any number outside of the ranges is in the gap.
2. Further, the two ranges can be combined into a single branch using the logical OR operator.

PARTICIPATION ACTIVITY 1.9.4: NFL Jersey numbers.

In the National Football League (NFL), player positions have jersey numbers in specific ranges. Ex: An NFL wide receiver can only wear jersey numbers from 10 to 19 or 80 to 89. Select the if statement that explicitly detects the correct NFL jersey number ranges.

1) Linebacker: 40 to 59 or 90 to 99

- ```
  if (j_num >= 40 and j_num
  <= 59) or (j_num >= 90
  and j_num <= 99):
  ```
- ```
  if (j_num > 40 and j_num
  <= 59) or (j_num > 90 and
  j_num <= 99):
  ```
- ```
  if j_num >= 40 and j_num
  <= 99:
  ```


2) Tight end: 40 to 49 or 80 to 89
  ```

- `if (40 <= j_num <= 49) and (80 <= j_num <= 89):`
- `if (j_num >= 40 or j_num <= 49) and (j_num >= 80 or j_num <= 89):`

3) Defensive lineman: 50 to 79 or 90 to 99

- `if (40 <= j_num <= 49) or (80 <= j_num <= 89):`
- `if (j_num > 50 and j_num < 79) or (j_num > 90 and j_num < 99):`
- `if (j_num >= 49 and j_num =< 80) or (j_num >= 89 and j_num <= 100):`
- `if (j_num > 49 and j_num < 80) or (j_num > 89 and j_num < 100):`

4) Quarterback: 1 to 19

- `if j_num <= 19:`
- `if j_num > 0 and j_num < 20:`
- `if j_num > 0 or j_num < 20:`

CHALLENGE ACTIVITY 1.9.1: Enter the output of the branch expressions.

422102.2723990.qx3zqy7

1.10 Detecting multiple features with branches

Multiple distinct if statements

Sometimes the programmer has multiple if statements in sequence, which looks similar to a multi-branch if-else statement but has a very different meaning. Each if statement is independent, and thus more than one branch can execute, in contrast to the multi-branch if-else arrangement. @@@

PARTICIPATION ACTIVITY 1.10.1: Multiple distinct if statements.

PARTICIPATION ACTIVITY 1.10.2: If statements.

Determine the final value of num_boxes.

1)
```
num_boxes  = 0
num_apples = 9

if num_apples < 20:
    num_boxes = 3
if num_apples < 10:
    num_boxes = num_boxes
- 1
```

Check Show answer

2)
```
num_boxes  = 0
num_apples = 9

if num_apples < 10:
    if num_apples < 5:
        num_boxes = 1
    else:
        num_boxes = 2
elif num_apples < 20:
    num_boxes = num_boxes
+ 1
```


Check Show answer

CHALLENGE ACTIVITY 1.10.1: Enter the output for the multiple if-else branches.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.10.2: If-else statements.

422102.2723990.qx3zqy7

Nested if-else statements

A branch's statements can include any valid statements, including another if-else statement, which are known as ***nested if-else*** statements.

The below Python Tutor tool traces a Python program's execution. The Python Tutor tool is available at www.pythontutor.com.

PARTICIPATION ACTIVITY 1.10.3: Nested if-else

PARTICIPATION ACTIVITY 1.10.4: Nested if-else statements.

Determine the final value of sales_bonus given the initial values specified below.

```
if sales_type == 2:
    if sales_bonus < 5:
        sales_bonus = 10
    else:
        sales_bonus = sales_bonus + 2
else:
    sales_bonus = sales_bonus + 1
```


1) sales_type = 1; sales_bonus = 0;

- 0
- 1
- 10

2) sales_type = 2; sales_bonus = 4;

- 5
- 6
- 10

3) sales_type = 2; sales_bonus = 7;

- 8
- 9
- 10

1.11 Comparing data types and common errors

Comparing integers, strings, and floating-point types

The relational and equality operators work for integer, string, and floating-point built-in types.

Floating-point types should not be compared using the equality operators, due to the imprecise representation of floating-point numbers.

The operators can also be used for the string type. Strings are equal if they have the same number of characters and corresponding characters are identical. If string my_str = 'Tuesday', then (my_str == 'Tuesday') is True, while (my_str == 'tuesday') is False because T differs from t.

PARTICIPATION ACTIVITY 1.11.1: Comparing various types.

Which comparisons will not result in a syntax error AND consistently yield expected results? Variables have types denoted by their names.

1) my_int == 42

- OK
- Not OK

2) my_float == 3.14

- OK
- Not OK

3) my_string == 'Hello'

- OK

The types of the values being compared determines the meaning of a comparison. If both values are numbers, then the numbers are compared arithmetically (`5 < 2` is False). Comparisons that make no sense, such as `1 < 'abc'` result in a TypeError.

Comparison of values with the same type, like `5 < 2`, or `'abc' >= 'ABCDEF'`, depend on the types being compared.

- Numbers are arithmetically compared.
- Strings are compared by converting each character to a number value (ASCII or Unicode), and then comparing each character in order. Most string comparisons use equality operators "==" or "!=", as in `today == 'Friday'.`
- Lists and tuples are compared via an ordered comparison of every element in the sequence. Every element between the sequences must compare as equal for an equality operator to evaluate to True. Relational operators like < or > can also be used: The result is determined by the first mismatching elements in the sequences. For example, if x = [1, 5, 2] and y = [1, 4, 3], then evaluating `x < y` first evaluates that 1 and 1 match. The next elements do not match, so `5 < 4` is evaluated, which produces a value of False.
- Dictionaries are compared only with == and !=. To be equal, two dictionaries must have the same set of keys and the same corresponding value for each key.

PARTICIPATION ACTIVITY 1.11.2: Comparing various types.

1) Click the expression that is False.

- 5 <= 5.0
- 10 != 9.999999
- (4 + 1) != 5.0

2) Click the expression that is False.

- 'FRIDAY' == 'friday'
- '1' < '2'
- 'a' != 'b' < 'c'

3) Click the expression that is True.

- {'Henrik': '$25'} == {'Daniel': '$25'}
- (1,2,3) > (0,2,3)
- [1, 2, 3] >= ['1', '2', '3']

Common branching errors

A common error is to use = rather than == in an if-else expression, as in: `if numDogs = 9:`. In such cases, the interpreter should generate a syntax error.

Another common error is to use invalid character sequences like =>, !<, or <>, which are *not* valid operators.

PARTICIPATION ACTIVITY 1.11.3: Watch out for assignment in an if-else expression.

What is the final value of num_items? Write "Error" if the code results in an error.

1)
```
num_items = 3
if num_items == 3:
   num_items = num_items
+ 1
```

Check Show answer

2)
```
num_items = 3
if num_items = 10:
   num_items = num_items
+ 1
```

Check Show answer

3)
```
num_items = 3
if num_items > 10:
   num_items = num_items
+ 1
```

Check Show answer

CHALLENGE ACTIVITY 1.11.1: If-else statement: Fix errors.

422102.2723990.qx3zqy7

1.12 While loops

This section has been set as optional by your instructor.

While loop: Basics

A ***while loop*** is a construct that repeatedly executes an indented block of code (known as the ***loop body***) as long as the loop's expression is True. At the end of the loop body, execution goes back to the while loop statement and the loop expression is evaluated *again*. If the loop expression is True, the loop body is executed again. But, if the expression evaluates to False, then execution instead proceeds to below the loop body. Each execution of the loop body is called an ***iteration***, and looping is also called *iterating*.

Construct 1.12.1: While loop.

```
while expression:  # Loop expression
    # Loop body: Sub-statements to execute
    # if the loop expression evaluates to True

# Statements to execute after the expression evaluates to
False
```

PARTICIPATION ACTIVITY 1.12.1: While loop.

Animation content:

undefined

Animation captions:

1. When encountered, a while loop's expression is evaluated. If true, the loop's body is entered. Here, user_char was initialized with 'y', so user_char == 'y' is true.
2. Thus, the loop body is executed, which outputs curr_power's current value of 2, doubles curr_power, and gets the next input.
3. Execution jumps back to the while part. user_char is 'y' (the first input), so user_char == 'y' is true, and the loop body executes (again), outputting 4.
4. user_char is 'y' (the second user input), so user_char == 'y' is true, and the loop body

executes (a third time), outputting 8.

5. user_char is now 'n', so user_char == 'y' is false. Thus, execution jumps to after the loop, which outputs "Done".

PARTICIPATION ACTIVITY 1.12.2: Basic while loops.

For each question, indicate how many times will the loop body execute?

1)
```
x = 3
while x >= 1:
    # Do something
    x = x - 1
```

Check Show answer

2) Assume user would enter 'n', then 'n', then 'y'.

```
# Get character from user here
while user_char != 'n':
    # Do something
    # Get character from user here
```

Check Show answer

3) Assume user would enter 'a', then 'b', then 'n'.

```
# Get character from user here
while user_char != 'n':
    # Do something
    # Get character from user here
```


Check Show answer

Example: While loop with a sentinel value

The following example uses the statement `while user_value != 'q':` to allow a user to end a face-drawing program by entering the character 'q'. The letter 'q' in this case is a ***sentinel value***, a value that when evaluated by the loop expression causes the loop to terminate.

The code `print(user_value*5)` produces a new string, which repeats the value of user_value 5 times. In this case, the value of user_value may be "-", thus the result of the multiplication is "-----". Another valid (but long and visually unappealing) method is the statement `print(f'{user_value}{user_value}{user_value}{user_value}{user_value}')`.

Note that `input` may read in a multi-character string from the user, so only the first character is extracted from user_input with `user_value = user_input[0]`.

Once execution enters the loop body, execution continues to the body's end even if the expression becomes False midway through.

Figure 1.12.1: While loop example: Face-printing program that ends when user enters 'q'.

```
nose = '0'  # Looks a little like a nose
user_value = '-'

while user_value != 'q':
    print(f' {user_value} {user_value} ')  # Print eyes
    print(f'  {nose}  ')  # Print nose
    print(user_value*5)  # Print mouth
    print('\n')

    # Get new character for eyes and mouth
    user_input = input("Enter a character ('q' for quit): \n")
    user_value = user_input[0]

print('Goodbye.\n')
```

```
 - -
  0
-----

Enter a character ('q'
for quit): x
 x x
  0
xxxxx


Enter a character ('q'
for quit): @
 @ @
  0
@@@@@


Enter a character ('q'
for quit): q
Goodbye.
```

PARTICIPATION ACTIVITY 1.12.3: Loop expressions.

Complete the loop expressions, using a single operator in your expression. Use the most straightforward translation of English to an expression.

1) Iterate while x is less than 100.

```
while
    :
    # Loop body statements
go here
```

Check Show answer

2) Iterate while x is greater than or equal to 0.


```
while
    :
    # Loop body statements
go here
```

Check Show answer

3) Iterate while c equals 'g'.

```
while
    :
    # Loop body statements
go here
```

Check Show answer

4) Iterate *until* c equals 'z'.

```
while
    :
    # Loop body statements
go here
```

Check Show answer

Stepping through a while loop

The following program animation provides another loop example. First, the user enters an integer. Then, the loop prints each digit one at a time starting from the right, using "% 10" to get the rightmost digit and "// 10" to remove that digit. The loop is only entered while num is greater than 0; once num reaches 0, the loop will have printed all digits.

PARTICIPATION ACTIVITY 1.12.4: While loop step-by-step.

Animation content:

undefined

Animation captions:

1. User enters the number 902. The first iteration prints "2".
2. The second iteration prints "0".
3. The third iteration prints "9", so every digit has been printed. The loop condition is checked one more time, and since num is 0, the loop stops.

Example: While loop: Iterations

Each iteration of the program below prints one line with the year and the number of ancestors in that year. (Note: the program's output numbers are largely due to not considering breeding among distant relatives, but nevertheless, a person has many ancestors.)

The program checks for `year_considered >= user_year` rather than for `year_considered != user_year`, because year_considered might be reduced past user_year without equaling it, causing an infinite loop. An ***infinite loop*** is a loop that will always execute because the loop's expression is always True. A common error is to accidentally create an infinite loop by assuming equality will be reached. Good practice is to include greater than or less than along with equality in a loop expression to help avoid infinite loops.

A program with an infinite loop may print output excessively, or just seem to stall (if the loop contains no printing). A user can halt a program by pressing Control-C in the command prompt running the Python program. Alternatively, some IDEs have a "Stop" button.

zyDE 1.12.1: While loop example: Ancestors printing program.

Run the program below.

Load default ter

```
year_considered = 2020  # Year being considered
num_ancestors = 2  # Approx. ancestors in considered year
years_per_generation = 20  # Approx. years per generation

user_year = int(input('Enter a past year (neg. for B.C.): '))
print()

while year_considered >= user_year:
    print(f'Ancestors in {year_considered}: {num_ancestors}')

    num_ancestors = num_ancestors * 2
    year_considered = year_considered - years_per_generation

```


1945

Run

PARTICIPATION ACTIVITY 1.12.5: While loop iterations.

What is the output of the following code? (Use "IL" for infinite loops.)

1)
```
x = 0
while x > 0:
    print(x, end=' ')
    x = x - 1
print('Bye')
```


Check **Show answer**

2)
```
x = 5
y = 18
while y >= x:
    print(y, end=' ')
    y = y - x
```

Check Show answer

3)
```
x = 10
while x != 3:
    print(x, end=' ')
    x = x / 2
```

Check Show answer

4)
```
x = 1
y = 3
z = 5
while not (y < x < z):
    print(x, end=' ')
    x = x + 1
```

Check Show answer

CHALLENGE ACTIVITY 1.12.1: Enter the output of the while loop.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.12.2: Basic while loop with user input.

Write an expression that executes the loop body as long as the user enters a non-negative number.

Note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds and report "Program end never reached." The system doesn't print the test case that caused the reported message.

Sample outputs with inputs: 9 5 2 -1

```
Body
Body
Body
Done.
```


422102.2723990.qx3zqy7

```
1 user_num = int(input())
2 while ''' Your solution goes here ''':
3   print('Body')
4   user_num = int(input())
5
6 print('Done.')
```

Run

CHALLENGE ACTIVITY 1.12.3: Basic while loop expression.

Write a while loop that repeats while user_num ≥ 1. In each loop iteration, divide user_num by 2, then print user_num.

Sample output with input: 20

```
10.0
5.0
2.5
1.25
0.625
```

Note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds and report "Program end never reached." The system doesn't print the test case that caused the reported message.

422102.2723990.qx3zqy7

```
1 user_num = int(input())
2
3 ''' Your solution goes here '''
4
```


Run

1.13 For loops

This section has been set as optional by your instructor.

Survey

The following questions are part of a zyBooks survey to help us improve our content so we can offer the best experience for students. The survey can be taken anonymously and takes just 3-5 minutes. Please take a short moment to answer by clicking the following link.

Link: Student survey

Basics

A common programming task is to access all of the elements in a container. Ex: Printing every item in a list. A ***for loop*** statement loops over each element in a container one at a time, assigning a variable with the next element that can then be used in the loop body. The container in the for loop statement is typically a list, tuple, or string. Each iteration of the loop assigns the name given in the for loop statement with the next element in the container.

Construct 1.13.1

```
for variable in container:
    # Loop body: Sub-statements to execute
    # for each item in the container

# Statements to execute after the for loop is
complete
```

PARTICIPATION ACTIVITY 1.13.1: Iterating over a list using a for loop.

Animation content:

undefined

Animation captions:

1. The first iteration assigns the variable name with 'Bill' and prints 'Hi Bill!' to the screen.
2. The second iteration assigns the variable name with 'Nicole' and prints 'Hi Nicole!'.
3. The third iteration assigns the variable name with 'John' and prints 'Hi John!'.

The for loop above iterates over the list `['Bill', 'Nicole', 'John']`. The first iteration assigns the variable name with 'Bill', the second iteration assigns name with 'Nicole', and the final iteration assigns name with 'John'. For sequence types like lists and tuples, the assignment order follows the position of the elements in the container, starting with position 0 (the leftmost element) and continuing until the last element is reached.

Iterating over a dictionary using a for loop assigns the loop variable with the *keys* of the dictionary. The values can then be accessed using the key.

Figure 1.13.1: A for loop assigns the loop variable with a dictionary's keys.

```
channels = {
    'MTV': 35,
    'CNN': 28,
    'FOX': 11,
    'NBC': 4,
    'CBS': 12
}

for c in channels:
    print(f'{c} is on channel
{channels[c]}')
```

```
MTV is on channel
35
CNN is on channel
28
FOX is on channel
11
NBC is on channel
4
CBS is on channel
12
```


A for loop can also iterate over a string. Each iteration assigns the loop variable with the next character of the string. Strings are sequence types just like lists, so the behavior is identical (leftmost character first, then each following character).

Figure 1.13.2: Using a for loop to access each character of a string.

```
my_str = ''
for character in "Take me to the
moon.":
    my_str += character + '_'
print(my_str)
```

```
T_a_k_e_ _m_e_ _t_o_ _t_h_e_
_m_o_o_n_._
```

PARTICIPATION ACTIVITY 1.13.2: Creating for loops.

Complete the for loop statement by giving the loop variable and container.

1) Iterate over the given list using a variable called my_pet.

```
for [          ] in
['Scooter', 'Kobe',
'Bella']:
    # Loop body statements
```

Check Show answer

2) Iterate over the list my_prices using a variable called price.

```
for          :
    # Loop body statements
```

Check Show answer

3) Iterate the string '911' using a variable called number.

```
for          :
    # Loop body statements
```

Check Show answer

For loop examples

For loops can be used to perform some action during each loop iteration. A simple example would be printing the value, as above examples demonstrated. The program below uses an additional variable to sum list elements to calculate weekly revenue and an average daily revenue.

Figure 1.13.3: For loop example: Calculating shop revenue.

```
daily_revenues = [
    2356.23,  # Monday
    1800.12,  # Tuesday
    1792.50,  # Wednesday
    2058.10,  # Thursday
    1988.00,  # Friday
    2002.99,  # Saturday
    1890.75   # Sunday
]

total = 0
for day in daily_revenues:
    total += day

average = total / len(daily_revenues)

print(f'Weekly revenue: ${total:.2f}')
print(f'Daily average revenue:
${average:.2f}')
```



```
Weekly revenue: $13888.69
Daily average revenue:
$1984.10
```

A for loop may also iterate backwards over a sequence, starting at the last element and ending with the first element, by using the ***reversed()*** function to reverse the order of the elements.

Figure 1.13.4: For loop example: Looping over a sequence in reverse.

The following program first prints a list that is ordered alphabetically, then prints the same list in reverse order.

```
names = [
    'Biffle',
    'Bowyer',
    'Busch',
    'Gordon',
    'Patrick'
]

for name in names:
    print(name, '|', end=' ')

print('\nPrinting in
reverse:')
for name in reversed(names):
    print(name, '|', end=' ')
```



```
Biffle | Bowyer | Busch | Gordon | Patrick
|
Printing in reverse:
Patrick | Gordon | Busch | Bowyer | Biffle
|
```

PARTICIPATION ACTIVITY 1.13.3: For loops.

Fill in the missing code to perform the desired calculation.

1) Compute the average number of kids.

```
# Each list item is the
number of kids in a
family.
num_kids = [1, 1, 2, 2, 1,
4, 3, 1]

total = 0
for num in num_kids:
    total +=
```



```
average = total /
len(num_kids)
```

Check Show answer

2) Assign num_neg with the

number of below-freezing Celsius temperatures in the list.

```
temperatures = [30, 20, 2,
-5, -15, -8, -1, 0, 5, 35]

num_neg = 0
for temp in temperatures:
    if temp < 0:
```


Check Show answer

3) Print scores in order from highest to lowest. Note: List is pre-sorted from lowest to highest.

```
scores = [75, 77, 80, 85,
90, 95, 99]

for scr in
    :
    print(scr, end=' ')
```

Check Show answer

CHALLENGE ACTIVITY 1.13.1: Looping over strings, lists, and dictionaries.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 1.13.2: For loop: Printing a list

Write an expression to print each price in stock_prices.

Sample output with inputs: 34.62 76.30 85.05

```
$ 34.62
$ 76.30
$ 85.05
```

422102.2723990.qx3zqy7

```
# NOTE: The following statement converts the input into a list container
stock_prices = input().split()

for ''' Your solution goes here ''':
    print('$', price)
```


Run

CHALLENGE ACTIVITY | 1.13.3: For loop: Printing a dictionary

Write a for loop to print each contact in contact_emails.

Sample output with inputs: 'Alf' 'alf1@hmail.com'

```
mike.filt@bmail.com is Mike Filt
s.reyn@email.com is Sue Reyn
narty042@nmail.com is Nate Arty
alf1@hmail.com is Alf
```

422102.2723990.qx3zqy7

```
contact_emails = {
    'Sue Reyn' : 's.reyn@email.com',
    'Mike Filt': 'mike.filt@bmail.com',
    'Nate Arty': 'narty042@nmail.com'
}

new_contact = input()
new_email = input()
contact_emails[new_contact] = new_email

```

Run

1.14 List basics

Creating a list

A ***container*** is a construct used to group related values together and contains references to other objects instead of data. A ***list*** is a container created by surrounding a sequence of variables or literals with brackets [ ]. Ex: `my_list = [10, 'abc']` creates a new list variable my_list that contains the two items: 10 and 'abc'. A list item is called an ***element*** .

A list is also a sequence, meaning the contained elements are ordered by position in the list, known as the element's ***index***, starting with 0. `my_list = [ ]` creates an empty list.

The animation below shows how a list is created and managed by the interpreter. A list itself is an object, and its value is a sequence of references to the list's elements.

PARTICIPATION ACTIVITY 1.14.1: Creating lists.

Animation captions:

1. User creates a new list.
2. The interpreter creates new object for each list element.
3. 'prices' holds references to objects in list.

zyDE 1.14.1: Creating lists.

The following program prints a list of names. Try adding your name to the list, and ru program again.

Load default template... **Run**


```
1 names = ['Daniel', 'Roxanna', 'Jean']
2
3 print(names)
4
```

PARTICIPATION ACTIVITY 1.14.2: Creating lists.

1) Write a statement that creates a list called my_nums, containing the elements 5, 10, and 20.

Check **Show answer**

2) Write a statement that creates a list called my_list with the elements -100 and the string 'lists are fun'.

Check **Show answer**

3) Write a statement that creates

an empty list called class_grades.

Check Show answer

Accessing list elements

Lists are useful for reducing the number of variables in a program. Instead of having a separate variable for the name of every student in a class, or for every word in an email, a single list can store an entire collection of related variables.

Individual list elements can be accessed using an indexing expression by using brackets as in my_list[i], where i is an integer. This allows a programmer to quickly find the i'th element in a list.

A list's index must be an integer. The index cannot be a floating-point type, even if the value is a whole number like 0.0 or 1.0. Using any type besides an integer will produce a runtime error and the program will terminate.

Figure 1.14.1: Access list elements using an indexing expression.

```
# Some of the most expensive cars in the world
lamborghini_veneno = 3900000  # $3.9 million!
bugatti_veyron = 2400000      # $2.4 million!
aston_martin_one77 = 1850000  # $1.85 million!

prices = [lamborghini_veneno, bugatti_veyron, aston_martin_one77]

print('Lamborghini Veneno:', prices[0], 'dollars')
print('Bugatti Veyron Super Sport:', prices[1], 'dollars')
print('Aston Martin One-77:', prices[2], 'dollars')
```

```
Lamborghini Veneno: 3900000 dollars
Bugatti Veyron Super Sport: 2400000 dollars
Aston Martin One-77: 1850000 dollars
```


CHALLENGE ACTIVITY 1.14.1: Initialize a list.

Initialize the list short_names with strings 'Gus', 'Bob', and 'Zoe'. Sample output for the given program:

```
Gus
Bob
Zoe
```


422102.2723990.qx3zqy7

```
1 short_names = ''' Your solution goes here '''
2
3 print(short_names[0])
4 print(short_names[1])
5 print(short_names[2])
```

Run

View your last submission

Updating list elements

Lists are mutable, meaning that a programmer can change a list's contents. An element can be updated with a new value by performing an assignment to a position in the list.

Figure 1.14.2: Updating list elements.

```
my_nums = [5, 12, 20]
print(my_nums)

# Update a list element
my_nums[1] = -28
print(my_nums)
```

```
[5, 12, 20]
[5, -28, 20]
```


PARTICIPATION ACTIVITY 1.14.3: Accessing and updating list elements.

1) Write a statement that assigns my_var with the 3rd element of my_list.

Check Show answer

2) Write a statement that assigns the 2nd element of my_towns with 'Detroit'.

Check Show answer

Adding and removing list elements

Since lists are mutable, a programmer can also use methods to add and remove elements. A ***method*** instructs an object to perform some action, and is executed by specifying the method name following a "." symbol and an object. The ***append()*** list method is used to add new elements to a list. Elements can be removed using the ***pop()*** or ***remove()*** methods. Methods are covered in greater detail in another section.

Adding elements to a list:

- list.append(value): Adds value to the end of list. Ex: `my_list.append('abc')`

Removing elements from a list:

- list.pop(i): Removes the element at index i from list. Ex: `my_list.pop(1)`
- list.remove(v): Removes the first element whose value is v. Ex: `my_list.remove('abc')`

PARTICIPATION ACTIVITY 1.14.4: Adding and removing list elements.

Animation content:

`undefined`

Animation captions:

1. append() adds an element to the end of the list.
2. pop() removes the element at the given index from the list. 'bw', which is at index 1, is removed and 'abc' is now at index 1.
3. remove() removes the first element with a given value. 'abc' is removed and now the list only has one element.

PARTICIPATION ACTIVITY 1.14.5: List modification.

Write a statement that performs the desired action. Assume the list `house_prices = ['$140,000', '$550,000', '$480,000']` exists.

1) Update the price of the second item in house_prices to '$175,000'.

Check Show answer

2) Add a price to the end of the list with a value of '$1,000,000'.

Check Show answer

3) Remove the 1st element from house_prices, using the pop() method.

Check Show answer

4) Remove '$140,000' from house_prices, using the remove() method.

Check Show answer

Sequence-type methods and functions

Sequence-type functions are built-in functions that operate on sequences like lists and strings. ***Sequence-type methods*** are methods built into the class definitions of sequences like lists and strings. A subset of such functions and methods is provided below.

Table 1.14.1: Some of the functions and methods useful to lists.

Operation	Description
len(list)	Find the length of the list.
list1 + list2	Produce a new list by concatenating list2 to the end of list1.
min(list)	Find the element in list with the smallest value. All elements must be of the same type.
max(list)	Find the element in list with the largest value. All elements must be of the same type.
sum(list)	Find the sum of all elements of a list (numbers only).
list.index(val)	Find the index of the first element in list whose value matches val.
list.count(val)	Count the number of occurrences of the value val in list.

Figure 1.14.3: Using sequence-type functions with lists.

```
# Concatenating lists
house_prices = [380000, 900000, 875000] +
[225000]
print('There are', len(house_prices), 'prices
in the list.')

# Finding min, max
print('Cheapest house:', min(house_prices))
print('Most expensive house:',
max(house_prices))
```

```
There are 4 prices in
the list.
Cheapest house: 225000
Most expensive house:
900000
```


Note that lists can contain mixed types of objects. Ex: `x = [1, 2.5, 'abc']` creates a new list x that contains an integer, a floating-point number, and a string. Later material explores lists in detail, including how lists can even contain other lists as elements.

zyDE 1.14.2: Student grade statistics.

The following program calculates some information regarding final and midterm sco
enhancing the program by calculating the average midterm and final scores.

Load default ter


```
#Program to calculate statistics from student test scores.
midterm_scores = [99.5, 78.25, 76, 58.5, 100, 87.5, 91, 68, 100]
final_scores = [55, 62, 100, 98.75, 80, 76.5, 85.25]

#Combine the scores into a single list
all_scores = midterm_scores + final_scores

num_midterm_scores = len(midterm_scores)
num_final_scores = len(final_scores)

print(num_midterm_scores, 'students took the midterm.')
print(num_final_scores, 'students took the final.')

#Calculate the number of students that took the midterm but not the final
dropped_students = num_midterm_scores - num_final_scores
print(dropped_students, 'students must have dropped the class.')

```

Run

PARTICIPATION ACTIVITY 1.14.6: Using sequence-type functions.

1) Write an expression that concatenates the list feb_temps to the end of jan_temps.

Check **Show answer**

2) Write an expression that finds the minimum value in the list total_prices.

Check Show answer

3) Write a statement that assigns the variable avg_price with the average of the elements of prices.

Check Show answer

CHALLENGE ACTIVITY 1.14.2: List functions and methods.

422102.2723990.qx3zqy7

1.15 Tuple basics

Tuples

A ***tuple***, usually pronounced "tuhple" or "toople", behaves similar to a list but is immutable – once created the tuple's elements cannot be changed. A tuple is also a sequence type, supporting len(), indexing, and other sequence type functions. A new tuple is generated by creating a list of comma-separated values, such as `5, 15, 20`. Typically, tuples are surrounded with parentheses, as in `(5, 15, 20)`. Note that printing a tuple always displays surrounding parentheses.

A tuple is not as common as a list in practical usage, but can be useful when a programmer wants to ensure that values do not change. Tuples are typically used when element position, and not just the relative ordering of elements, is important. Ex: A tuple might store the latitude and longitude of a landmark because a programmer knows that the first element should be the latitude, the second element should be the longitude, and the landmark will never move from those coordinates.

Figure 1.15.1: Using tuples.

```
white_house_coordinates = (38.8977,
77.0366)
print('Coordinates:',
white_house_coordinates)
print('Tuple length:',
len(white_house_coordinates))

# Access tuples via index
print('\nLatitude:',
white_house_coordinates[0], 'north')
print('Longitude:',
white_house_coordinates[1], 'west\n')

# Error. Tuples are immutable
white_house_coordinates[1] = 50
```

```
Coordinates: (38.8977, 77.0366)
Tuple length: 2

Latitude: 38.8977 north
Longitude: 77.0366 west

Traceback (most recent call
last):
  File "<stdin>", line 10, in
<module>
TypeError: 'tuple' object does
not support item assignment
```


PARTICIPATION ACTIVITY 1.15.1: Tuples.

1) Create a new variable `point` that is a tuple containing the strings `'X string'` and `'Y string'`.

Check Show answer

2) If the value of variable `friends` is the tuple `('Cleopatra', 'Marc', 'Seneca')`, then what is the result of `len(friends)`?

Check Show answer

CHALLENGE ACTIVITY 1.15.1: Initialize a tuple.

Initialize the tuple `team_names` with the strings 'Rockets', 'Raptors', 'Warriors', and 'Celtics' (The top-4 2018 NBA teams at the end of the regular season in order). Sample output for the given program:

```
Rockets
Raptors
Warriors
Celtics
```


422102.2723990.qx3zqy7

```
1 team_names = ''' Your solution goes here '''
2
3 print(team_names[0])
4 print(team_names[1])
5 print(team_names[2])
6 print(team_names[3])
```

Run

View your last submission

Named tuples

A program commonly captures collections of data; for example, a car could be described using a series of variables describing the make, model, retail price, horsepower, and number of seats. A ***named tuple*** allows the programmer to define a new simple data type that consists of named attributes. A `Car` named tuple with fields like `Car.price` and `Car.horsepower` would more clearly represent a car object than a list with index positions correlating to some attributes.

The ***namedtuple*** container must be imported to create a new named tuple. Once the container is imported, the named tuple should be created like in the example below, where the name and attribute names of the named tuple are provided as arguments to the namedtuple constructor. Note that the fields to include in the named tuple are found in a list, but may also be a single string

with space or comma separated values.

Figure 1.15.2: Creating named tuples.

```
from collections import namedtuple

Car = namedtuple('Car', ['make','model','price','horsepower','seats'])
# Create the named tuple

chevy_blazer = Car('Chevrolet', 'Blazer', 32000, 275, 8)  # Use the
named tuple to describe a car
chevy_impala = Car('Chevrolet', 'Impala', 37495, 305, 5)  # Use the
named tuple to describe a different car

print(chevy_blazer)
print(chevy_impala)
```

```
Car(make='Chevrolet', model='Blazer', price=32000, horsepower=275, seats=8)
Car(make='Chevrolet', model='Impala', price=37495, horsepower=305, seats=5)
```


`namedtuple()` only creates the new simple data type, and does not create new data objects. Above, a new data object is not created until `Car()` is called with appropriate values. A data object's attributes can be accessed using dot notation, as in `chevy_blazer.price`. This "named" attribute is simpler to read than if using a list or tuple referenced via index like `chevy_blazer[2]`.

Like normal tuples, named tuples are immutable. A programmer wishing to edit a named tuple would replace the named tuple with a new object.

PARTICIPATION ACTIVITY 1.15.2: Named tuples.

Assume namedtuple has been imported. Use a list of strings in the `namedtuple()` constructor where applicable.

1) Complete the following named tuple definition that describes a house.

```
House =
```

```
('House', ['street',
'postal_code', 'country'])
```

Check Show answer

2) Create a new named tuple `Dog` that has the attributes `name`, `breed`, and `color`.

Check Show answer

3) Let `Address = namedtuple('Address', ['street', 'city', 'country'])`. Create a new address object `house` where `house.street` is "221B Baker Street", `house.city` is "London", and `house.country` is "England".

Check Show answer

4) Given the following named tuple `Car = namedtuple('Car', ['make', 'model', 'price', 'horsepower', 'seats'])`, and data objects `car1` and `car2`, write an expression that computes the sum of the price of both cars.

Check Show answer

CHALLENGE ACTIVITY 1.15.2: Creating a named tuple

Define a named tuple `Player` that describes an athlete on a sports team. Include the fields `name`, `number`, `position`, and `team`.

422102.2723990.qx3zqy7

```
1 from collections import namedtuple
2
3 Player = ''' Your solution goes here '''
4
5 cam = Player('Cam Newton', '1', 'Quarterback', 'Carolina Panthers')
6 lebron = Player('Lebron James', '23', 'Small forward', 'Los Angeles Lakers')
```

Run

View your last submission

1.16 Set basics

Set basics

A ***set*** is an unordered collection of unique elements. Sets have the following properties:

- Elements are unordered: Elements in the set do not have a position or index.
- Elements are unique: No elements in the set share the same value.

A set can be created using the ***set()*** function, which accepts a sequence-type iterable object (list, tuple, string, etc.) whose elements are inserted into the set. A ***set literal*** can be written using curly braces `{ }` with commas separating set elements. Note that an empty set can only be created using `set()`.

Figure 1.16.1: Creating sets.

```
# Create a set using the set()
function.
nums1 = set([1, 2, 3])

# Create a set using a set literal.
nums2 = { 7, 8, 9 }

# Print the contents of the sets.
print(nums1)
print(nums2)
```

```
{1, 2,
3}
{7, 8,
9}
```


Because the elements of a set are unordered and have no meaningful position in the collection, the index operator is not valid. Attempting to access the element of a set by position, for example `nums1[2]` to access the element at index 2, is invalid and will produce a runtime error.

A set is often used to reduce a list of items that potentially contains duplicates into a collection of unique values. Simply passing a list into `set()` will cause any duplicates to be omitted in the created set.

zyDE 1.16.1: Creating sets.

Load default template... Run

```
1 # Initial list contains some duplicate value
2 first_names = [ 'Alba', 'Hema', 'Ron', 'Alba
3
4 # Creating a set removes any duplicate value
5 names_set = set(first_names)
6
7 print(names_set)
8
```

PARTICIPATION ACTIVITY 1.16.1: Basic sets.

1) What's the result of `set(['A', 'Z'])`?
   - A set that contains `'A'` and `'Z'`.
   - A list with the following elements: `['A', 'Z']`.
   - Error: invalid syntax.

2) What's the result of `set(10, 20, 25)`?
   - A list with the following elements: `[10, 20, 25]`.
   - A set that contains `10`, `20`, and `25`.
   - Error: invalid syntax.

3) What's the result of `set([100, 200, 100, 200, 300])`?
   - A list with the following elements: `[100, 200, 100, 200, 300]`.
   - A set that contains `100`, `200`, and `300`.
   - A set that contains `100`, `200`, `300`, another `100`, and another `200`.
   - Error: invalid syntax.

Modifying sets

Sets are mutable – elements can be added or removed using set methods. The ***add()*** method places a new element into the set if the set does not contain an element with the provided value. The ***remove()*** and ***pop()*** methods remove an element from the set.

Additionally, sets support the `len()` function to return the number of elements in a set. To check if a specific value exists in a set, a membership test such as `value in set` (discussed in another section) can be used.

Adding elements to a set:

- set.add(value): Add value into the set. Ex: `my_set.add('abc')`

Remove elements from a set:

- set.remove(value): Remove the element with given value from the set. Raises KeyError if value is not found. Ex: `my_set.remove('abc')`
- set.pop(): Remove a random element from the set. Ex: `my_set.pop()`

Table 1.16.1: Some of the methods useful to sets.

Operation	Description
len(set)	Find the length (number of elements) of the set.
set1.update(set2)	Adds the elements in set2 to set1.
set.add(value)	Adds value into the set.
set.remove(value)	Removes value from the set. Raises KeyError if value is not found.
set.pop()	Removes a random element from the set.
set.clear()	Clears all elements from the set.

PARTICIPATION ACTIVITY 1.16.2: Modifying sets.

Animation content:

Animation captions:

1. Sets can be created using braces {} with commas separating the elements.
2. The add() method adds a single element to a set.
3. The update() method adds the elements of one set to another set.
4. The remove() method removes a single element from a set.
5. The clear() method removes all elements from a set, leaving the set with a length of 0.

PARTICIPATION ACTIVITY 1.16.3: Modifying sets.

Write a line of code to complete the following operations.

1) Add the literal `'Ryder'` to the set `names`.

Check Show answer

2) Add all of the elements of set `goblins` into set `monsters`.

Check Show answer

3) Remove all of the elements from the `trolls` set.

Check Show answer

4) Get the number of elements in the set `elves`.

Check Show answer

CHALLENGE ACTIVITY 1.16.1: Creating and modifying sets.

The top three most popular male names of 2017 are `Oliver`, `Declan`, and `Henry`, according to babynames.com.

Write a program that modifies the `male_names` set by removing a name and adding a different name.

Sample output with inputs: 'Oliver' 'Atlas'

```
{ 'Atlas', 'Declan', 'Henry' }
```

NOTE: Because sets are unordered, the order in which the names in `male_names` appear may differ from above.

422102.2723990.qx3zqy7

```
male_names = { 'Oliver', 'Declan', 'Henry' }
name_to_remove = input()
name_to_add = input()

''' Your solution goes here '''

print(male_names)
```

Run

Set operations

Python set objects support typical set theory operations like intersections and unions. A brief overview of common set operations supported in Python are provided below:

Table 1.16.2: Common set theory operations.

.

Operation	Description
set.intersection(set_a, set_b, set_c...)	Returns a new set containing only the elements in common between `set` and all provided sets.
set.union(set_a, set_b, set_c...)	Returns a new set containing all of the unique elements in all sets.
set.difference(set_a, set_b, set_c...)	Returns a set containing only the elements of `set` that are not found in any of the provided sets.
set_a.symmetric_difference(set_b)	Returns a set containing only elements that appear in exactly one of `set_a` or `set_b`

PARTICIPATION ACTIVITY 1.16.4: Set theory operations.

Animation content:

Animation captions:

1. The union() method builds a set containing the unique elements from names1 and names2. 'Corrin' only appears once in the resulting set.
2. The intersection() method builds a set that contains all common elements between result_set and names3.
3. The difference() method builds a set that contains elements only found in result_set that are not in names4.

PARTICIPATION ACTIVITY 1.16.5: Set theory operations.

Assume that:

- `monsters = {'Gorgon', 'Medusa'}`
- `trolls = {'William', 'Bert', 'Tom'}`

- `horde = {'Gorgon', 'Bert', 'Tom'}`

Fill in the code to complete the line that would produce the given set.

1) `{'Gorgon', 'Bert', 'Tom', 'Medusa', 'William'}`

```
monsters.        (trolls)
```

Check Show answer

2) `{'Gorgon'}`

```
monsters.
(horde)
```

Check Show answer

3) `{'Medusa', 'Bert', 'Tom'}`

```
monsters.symmetric_difference(
)
```

Check Show answer

CHALLENGE ACTIVITY 1.16.2: Set theory methods.

The following program includes 10 cities that two people have visited. Write a program that creates:

1. A set `all_cities` that contains all of the cities both people have visited.
2. A set `same_cities` that contains only cities found in both `person1_cities` and `person2_cities`.
3. A set `different_cities` that contains cities found only in `person1_cities` or only in `person2_cities`.

Sample output for `all_cities`:

```
['Accra', 'Anaheim', 'Bangkok', 'Bend', 'Boise', 'Buenos Aires',
'Cairo', 'Edmonton', 'Lima', 'London', 'Memphis', 'Orlando',
'Paris', 'Seoul', 'Tokyo', 'Vancouver', 'Zurich']
```

NOTE: Because sets are unordered, they are printed using the sorted() function here for

comparison.

422102.2723990.qx3zqy7

```
person1_cities = {'Edmonton', 'Vancouver', 'Paris', 'Bangkok', 'Bend', 'Boise', 'Memphis'
person2_cities = {'Accra', 'Orlando', 'Tokyo', 'Paris', 'Anaheim', 'Buenos Aires', 'Londo

# Use set methods to create sets all_cities, same_cities, and different_cities.

''' Your solution goes here '''

print(sorted(all_cities))
print(sorted(same_cities))
print(sorted(different_cities))
```


Run

1.17 Dictionary basics

Creating a dictionary

Consider a normal English language dictionary – a reader looks up the word "cat" and finds the definition, "A small, domesticated carnivore." The relationship between "cat" and its definition is *associative*, i.e., "cat" is associated with some words describing "cat."

A ***dictionary*** is a Python container used to describe associative relationships. A dictionary is represented by the ***dict*** object type. A dictionary associates (or "maps") keys with values. A ***key*** is a term that can be located in a dictionary, such as the word "cat" in the English dictionary. A ***value*** describes some data associated with a key, such as a definition. A key can be any immutable type, such as a number, string, or tuple; a value can be any type.

A dict object is created using ***curly braces*** { } to surround the ***key:value pairs*** that comprise the dictionary contents. Ex: `players = {'Lionel Messi': 10, 'Cristiano Ronaldo': 7}` creates a dictionary called players with two keys: 'Lionel Messi' and 'Cristiano Ronaldo', associated with the values 10 and 7 (their respective jersey numbers). An empty dictionary is created with the expression `players = { }`.

Dictionaries are typically used in place of lists when an associative relationship exists. Ex: If a

program contains a collection of anonymous student test scores, those scores should be stored in a list. However, if each score is associated with a student name, a dictionary could be used to associate student names to their score. Other examples of associative relationships include last names and addresses, car models and price, or student ID number and university email address.

Figure 1.17.1: Creating a dictionary.


```
players = {
    'Lionel Messi': 10,
    'Cristiano Ronaldo':
7
}

print(players)
```

```
{'Lionel Messi': 10, 'Cristiano Ronaldo':
7}
```

Note that formatting list or dictionary entries like in the above example, where elements appear on consecutive lines, helps to improve the readability of the code. The behavior of the code is not changed.

zyDE 1.17.1: Creating dictionaries.

Run the program below that displays the caffeine content in milligrams for 100 ml/gr some popular foods. The indentation and spacing of the caffeine_content_mg key-va simply provides more readability. Note that order *is* maintained in the dict when print standard before Python 3.7).

Try adding new items into the dictionary, using this U.S. federal government report or content.

Load default template... **Run**

```
caffeine_content_mg = {
    'Mr. Goodbar chocolate': 122,
    'Red Bull': 33,
    'Monster Hitman Sniper energy drink': 7
    'Lipton Brisk iced tea - lemon flavor'
    'dark chocolate coated coffee beans': 8
    'Regular drip or percolated coffee': 60
    'Buzz Bites Chocolate Chews': 1639
}

print(caffeine_content_mg)

```

PARTICIPATION ACTIVITY 1.17.1: Create a dictionary.

1) Use braces to create a dictionary called ages that maps the names 'Bob' and 'Frank' to their ages, 27 and 75, respectively. For this exercise, make 'Bob' the first entry in the dict.


```
ages =
```

Check Show answer

Accessing dictionary entries

Though dictionaries maintain a left-to-right ordering, dictionary entries cannot be accessed by indexing. To access an entry, the key is specified in brackets [ ]. If no entry with a matching key exists in the dictionary, then a ***KeyError*** runtime error occurs and the program is terminated.

Figure 1.17.2: Accessing dictionary entries.


```
prices = {'apples': 1.99, 'oranges': 1.49}

print(f'The price of apples is
{prices["apples"]}')
print(f'\nThe price of lemons is
{prices["lemons"]}')
```

```
The price of apples is 1.99
Traceback (most recent call
last):
  File "<stdin>", line 3, in
<module>
KeyError: 'lemons'
```

PARTICIPATION ACTIVITY 1.17.2: Accessing dictionary entries.

1) A dictionary entry is accessed by placing a key in curly braces { }.
   - True
   - False

2) Dictionary entries are ordered by position.
   - True
   - False

Adding, modifying, and removing dictionary entries

A dictionary is mutable, so entries can be added, modified, and deleted as necessary by a programmer. A new dictionary entry is added by using brackets to specify the key: `prices['banana'] = 1.49`. A dictionary key is unique – attempting to create a new entry with a key that already exists in the dictionary *replaces* the existing entry. The ***del*** keyword is used to remove entries from a dictionary: `del prices['papaya']` removes the entry whose key is 'papaya'. If the requested key to delete does not exist then a KeyError occurs.

Adding new entries to a dictionary:

- dict[k] = v: Adds the new key-value pair k-v, if dict[k] does not already exist.
  Example: `students['John'] = 'A+'`

Modifying existing entries in a dictionary:

- dict[k] = v: Updates the existing entry dict[k], if dict[k] already exists.
  Example: `students['Jessica'] = 'A+'`

Removing entries from a dictionary:

- del dict[k]: Deletes the entry dict[k].
  Example: `del students['Rachel']`

Figure 1.17.3: Adding and editing dictionary entries.

```
prices = {}  # Create empty
dictionary
prices['banana'] = 1.49  # Add new
entry
print(prices)

prices['banana'] = 1.69  # Modify
entry
print(prices)

del prices['banana']  # Remove entry
print(prices)
```

```
{'banana':
1.49}
{'banana':
1.69}
{}
```

PARTICIPATION ACTIVITY 1.17.3: Modifying dictionaries.

1) Which statement adds 'pears' to the following dictionary?

```
prices = {'apples': 1.99,
'oranges': 1.49, 'kiwi': 0.79}
```

- prices['pears'] = 1.79
- prices['pears': 1.79]

2) Executing the following statements produces a KeyError:

```
prices = {'apples': 1.99,
'oranges': 1.49, 'kiwi': 0.79}
del prices['limes']
```

- True
- False

3) Executing the following statements adds a new entry to the dictionary:

```
prices = {'apples': 1.99,
'oranges': 1.49, 'kiwi': 0.79}
prices['oranges'] = 1.29
```


- True
- False

CHALLENGE ACTIVITY 1.17.1: Modify and add to dictionary.

Write a statement to add the key Tesla with value USA to car_makers. Modify the car maker of Fiat to Italy. Sample output for the given program:

```
Acura made in Japan
Fiat made in Italy
Tesla made in USA
```

422102.2723990.qx3zqy7

```
1  car_makers = {'Acura': 'Japan', 'Fiat': 'Egypt'}
2
3  # Add the key Tesla with value USA to car_makers
4  # Modify the car maker of Fiat to Italy
5
6  ''' Your solution goes here '''
7
8  print('Acura made in', car_makers['Acura'])
9  print('Fiat made in', car_makers['Fiat'])
10 print('Tesla made in', car_makers['Tesla'])
```


Run

1.18 Common data types summary

The most common Python types are presented below.

Common data types

Numeric types int and float represent the most common types used to store data. All numeric types support the normal mathematical operations such as addition, subtraction, multiplication, and division, among others.

Table 1.18.1: Common data types.

Type	Notes
int	Numeric type: Used for variable-width integers.
float	Numeric type: Used for floating-point numbers.

Sequence types string, list, and tuple are all containers for collections of objects ordered by position in the sequence, where the first object has an index of 0 and subsequent elements have indices 1, 2, etc. A list and a tuple are very similar, except that a list is mutable and individual elements may be edited or removed. Conversely, a tuple is immutable and individual elements may not be edited or removed. Lists and tuples can contain any type, whereas a string contains only single-characters. Sequence-type functions such as len() and element indexing using brackets [ ] can be applied to any sequence type.

The only ***mapping type*** in Python is the dict type. Like a sequence type, a dict serves as a container. However, each element of a dict is independent, having no special ordering or relation to other elements. A dictionary uses key-value pairs to associate a key with a value.

Table 1.18.2: Containers: sequence and mapping types.

Type	Notes
string	Sequence type: Used for text.
list	Sequence type: A mutable container with ordered elements.
tuple	Sequence type: An immutable container with ordered elements.
set	Set type: A mutable container with unordered and unique elements.
dict	Mapping type: A container with key-values associated elements.

PARTICIPATION ACTIVITY 1.18.1: Common data types.

1) The list `['a', 'b', 3]` is invalid because the list contains a mix of strings and integers.
   - True
   - False

2) int and float types can always hold the exact same values.
   - True
   - False

3) A sorted collection of integers might best be contained in a list.
   - True
   - False

Choosing a container type

New programmers often struggle with choosing the types that best fit their needs, such as

choosing whether to store particular data using a list, tuple, or dict. In general, a programmer might use a list when data has an order, such as lines of text on a page. A programmer might use a tuple instead of a list if the contained data should not change. If order is not important, a programmer might use a dictionary to capture relationships between elements, such as student names and grades.

PARTICIPATION ACTIVITY 1.18.2: Choosing among different container types.

Choose the container that best fits the described data.

1) Student test scores that may later be adjusted, ordered from best to worst.
   - ○ list
   - ○ tuple
   - ○ dict

2) A single student's name and their final grade in the class.
   - ○ list
   - ○ tuple
   - ○ dict

3) Names and current grades for all students in the class.
   - ○ list
   - ○ tuple
   - ○ dict

PARTICIPATION ACTIVITY 1.18.3: Finding errors in container code.

Click on the error.

1)
```
# Student grade program

students = ['Jo', 'Bob',
           'Amy']
```

```
grades = {}

# Get student name, grade
name = input('name:')
grade = input('grade:')# Assign
grade


   grades.append(name) =
             grade
```


2)

```
workers = ('Jo', 'Amy')

# Remove Amy from workers

del workers[1]

# Print worker at index 0

print('Jo:', workers[0])
```

1.19 Counting using the range() function

The range() function

While loops are commonly used for counting a specific number of iterations, and for loops are commonly used to iterate over all elements of a container. The range() function allows counting in for loops as well. ***range()*** generates a sequence of integers between a starting integer that is included in the range, an ending integer that is not included in the range, and an integer step value. The sequence is generated by starting at the start integer and incrementing by the step value until the ending integer is reached or surpassed.

The range() function can take up to three integer arguments.

- `range(Y)` generates a sequence of all non-negative integers less than Y.
  Ex: `range(3)` creates the sequence 0, 1, 2.
- `range(X, Y)` generates a sequence of all integers >= X and < Y.

Ex: `range(-7, -3)` creates the sequence -7, -6, -5, -4.

- `range(X, Y, Z)`, where Z is positive, generates a sequence of all integers >= X and < Y, incrementing by Z.
  Ex: `range(0, 50, 10)` creates the sequence 0, 10, 20, 30, 40.
- `range(X, Y, Z)`, where Z is negative, generates a sequence of all integers <= X and > Y, incrementing by Z.
  Ex: `range(3, -1, -1)` creates the sequence 3, 2, 1, 0.

Table 1.19.1: Using the range() function.

Range	Generated sequence	Explanation
`range(5)`	`0 1 2 3 4`	Every integer from 0 to 4.
`range(0, 5)`	`0 1 2 3 4`	Every integer from 0 to 4.
`range(3, 7)`	`3 4 5 6`	Every integer from 3 to 6.
`range(10, 13)`	`10 11 12`	Every integer from 10 to 12.
`range(0, 5, 1)`	`0 1 2 3 4`	Every 1 integer from 0 to 4.
`range(0, 5, 2)`	`0 2 4`	Every 2nd integer from 0 to 4.
`range(5, 0, -1)`	`5 4 3 2 1`	Every 1 integer from 5 down to 1
`range(5, 0, -2)`	`5 3 1`	Every 2nd integer from 5 down to 1

Evaluating the range() function creates a new "range" type object. Ranges represent an arithmetic progression, i.e., some sequence of integers with a start, end, and step between integers. The range type is a sequence type like lists and tuples, but is immutable. In general, range objects are only used as a part of a for loop statement.

zyDE 1.19.1: For loop example: Calculating yearly savings.

The below program uses a for loop to calculate savings and interest. Try changing th function to print every three years instead, using the three-argument alternate versior range(). Modify the interest calculation inside the loop to compute three years worth savings instead of one.

Load default template...

```
'''Program that calculates savings and inte

initial_savings = 10000
interest_rate = 0.05

years = int(input('Enter years: '))
print()

savings = initial_savings
for i in range(years):
    print(f' Savings in year {i}: ${savings
    savings = savings + (savings*interest_

print('\n')

```

8

Run

PARTICIPATION ACTIVITY 1.19.1: The range() function.

1) What sequence is generated by `range(7)`?

- 0 1 2 3 4 5 6 7
- 1 2 3 4 5 6
- 0 1 2 3 4 5 6

2) What sequence is generated by `range(2, 5)`?

- 2 3 4
- 2 3 4 5
- 0 1 2 3 4

PARTICIPATION ACTIVITY 1.19.2: The range() function.

Write the simplest range() function that generates the appropriate sequence of integers.

1) Every integer from 0 to 500.

Check Show answer

2) Every integer from 10 to 20

Check Show answer

3) Every 2nd integer from 10 to 20

Check Show answer

4) Every integer from 5 down to -5

Check Show answer

CHALLENGE ACTIVITY 1.19.1: Enter the for loop's output.

422102.2723990.qx3zqy7

1.20 Dynamic typing

Dynamic and static typing

A programmer can pass any type of object as an argument to a function. Consider a function add(x, y) that adds the two parameters:

A programmer can call the `add()` function using two integer arguments, as in `add(5, 7)`, which returns a value of 12. Alternatively, a programmer can pass in two string arguments, as in `add('Tora', 'Bora')`, which would concatenate the two strings and return `'ToraBora'`.

PARTICIPATION ACTIVITY 1.20.1: Polymorphic functions.

The function's behavior of being able to add together different types is a concept called ***polymorphism***. Polymorphism is an inherent part of the Python language. For example, consider the multiplication operator *. If the two operands are numbers, then the result is the product of those two numbers. If one operand is a string and the other an integer (e.g., `'x' * 5`) then the result is a repetition of the string 5 times: `'xxxxx'`.

Python uses ***dynamic typing*** to determine the type of objects as a program executes. Ex: The consecutive statements `num = 5` and `num = '7'` first assign with an integer type, and then a string type. The type of num can change, depending on the value it references. The interpreter is responsible for checking that all operations are valid as the program executes. If the function call `add(5, '100')` is evaluated, an error is generated when adding the string to an integer.

In contrast to dynamic typing, many other languages like C, C++, and Java use ***static typing***, which requires the programmer to define the type of every variable and every function parameter in a program's source code. Ex: `string name = "John"` would declare a string variable in C and C++. When the source code is compiled, the compiler attempts to detect non type-safe operations, and halts the compilation process if such an operation is found.

Dynamic typing typically allows for more flexibility in terms of the code that a programmer can write, but at the expense of potentially introducing more bugs, since there is no compilation

process by which types can be checked. [1]

PARTICIPATION ACTIVITY 1.20.2: Dynamic and static typing.

1) Polymorphism refers to how an operation depends on the involved object types.
   - True
   - False

2) A programmer can pass only string arguments to a user-defined function.
   - True
   - False

3) Static-typed languages require that the type of every variable is defined in the source code.
   - True
   - False

4) A dynamic-typed language like Python checks that an operation is valid when that operation is executed by the interpreter. If the operation is invalid, a run-time error occurs.
   - True
   - False

(*1) Python uses duck typing, a form of dynamic typing based on the maxim "If a bird walks, swims, and quacks like a duck, then call it a duck." For example, if an object can be concatenated, sliced, indexed, and converted to lower case, doing everything that a string can do, then treat the object like a string.

1.21 Functions: Common errors

Copy-paste errors

A common error is to copy-and-paste code among functions but then not complete all necessary modifications to the pasted code. For example, a programmer might have developed and tested a function to convert a temperature value in Celsius to Fahrenheit, and then copied and modified the original function into a new function to convert Fahrenheit to Celsius as shown:

Figure 1.21.1: Copy-paste common error: Pasted code not properly modified. Find error on the right.


```
def
celsius_to_fahrenheit(celsius):
    temperature = (9.0/5.0) *
celsius
    fahrenheit = temperature +
32

    return fahrenheit
```

```
def
fahrenheit_to_celsius(fahrenheit):
    temperature = fahrenheit- 32
    celsius = temperature *
(5.0/9.0)

    return fahrenheit
```

The programmer forgot to change the return statement to return `celsius` rather than `fahrenheit`. Copying-and-pasting code is a common and useful time-saver and can reduce errors by starting with known-correct code. Our advice is that when you copy-paste code, be extremely vigilant in making all necessary modifications. Just as the awareness that dark alleys or wet roads may be dangerous can cause you to vigilantly observe your surroundings or drive carefully, the awareness that copying-and-pasting is a common source of errors may cause you to more vigilantly ensure you modify a pasted function correctly.

PARTICIPATION ACTIVITY 1.21.1: Copy-pasted sum-of-squares code.

Original parameters were **num1**, **num2**, **num3**.
Original code was:

```
sum = (num1 * num1) + (num2 * num2) + (num3 * num3)
return sum
```


New parameters are **num1**, **num2**, **num3**, **num4**. Find the error in the copy-pasted new code below.

1) sum =
(num1 * num1) + (num2 * num2) + (num3 * num3) + (num3 * num4)

```
return sum
```

Return errors

Another common error is to return the wrong variable, like if `return temperature` had been used in the temperature conversion program by accident. The function will work and sometimes even return the correct value.

Another common error is to fail to return a value for a function. If execution reaches the end of a function's statements without encountering a return statement, then the function returns a value of `None`. If the function is expected to return an actual value, then such an assignment can cause confusion.

PARTICIPATION ACTIVITY 1.21.2: Missing return common error.

The program above produces unexpected output, leading to a bug that's hard to find. The program does not contain syntax errors, but does contain a logic error because the function

`steps_to_feet()` always returns a value `None`.

PARTICIPATION ACTIVITY 1.21.3: Common function errors.

1) Forgetting to return a value from a function is a common error.
   - True
   - False

2) Copying-and-pasting code can lead to common errors if all necessary changes are not made to the pasted code.
   - True
   - False

3) Returning the incorrect variable from a function is a common error.
   - True
   - False

4) Is this function correct for squaring an integer?

```
def sqr (a):
    t = a * a
```

   - Yes
   - No

5) Is this function correct for squaring an integer?

```
def sqr (a):
    t = a * a
    return a
```

   - Yes
   - No

CHALLENGE ACTIVITY 1.21.1: Function errors: Copying one function to create another.

Using the `celsius_to_kelvin` function as a guide, create a new function, changing the

name to `kelvin_to_celsius`, and modifying the function accordingly.

Sample output with input: 283.15

```
10.0 C is 283.15 K
283.15 K is 10.0 C
```


422102.2723990.qx3zqy7

```
1  def celsius_to_kelvin(value_celsius):
2      value_kelvin = 0.0
3
4      value_kelvin = value_celsius + 273.15
5      return value_kelvin
6
7  ''' Your solution goes here '''
8
9  value_c = 10.0
10 print(value_c, 'C is', celsius_to_kelvin(value_c), 'K')
11
12 value_k = float(input())
13 print(value_k, 'K is', kelvin_to_celsius(value_k), 'C')
```

Run

1.22 Function arguments

Function arguments and mutability

Arguments to functions are passed by object reference, a concept known in Python as ***pass-by-assignment***. When a function is called, new local variables are created in the function's local namespace by binding the names in the parameter list to the passed arguments.

PARTICIPATION ACTIVITY 1.22.1: Assignments to parameters have no effect outside the function.

Animation content:

```
undefined
```

Animation captions:

1. timmy_age and age reference the same object.
2. Assigning the parameter age with a new value doesn't change timmy_age.
3. Since timmy_age has not changed, "Timmy is 7" is displayed.

The semantics of passing object references as arguments is important because modifying an argument that is referenced elsewhere in the program may cause side effects outside of the function scope. When a function modifies a parameter, whether or not that modification is seen outside the scope of the function depends on the *mutability* of the argument object.

- If the object is **immutable**, such as a string or integer, then the modification is limited to inside the function. Any modification to an immutable object results in the creation of a *new* object in the function's local scope, thus leaving the original argument object unchanged.
- If the object is **mutable**, then in-place modification of the object can be seen outside the scope of the function. Any operation like adding elements to a container or sorting a list that is performed within a function will also affect any other variables in the program that reference the same object.

The following program illustrates how the modification of a list argument's elements inside a function persists outside of the function call.

PARTICIPATION ACTIVITY 1.22.2: Modification of a list inside a function.

Sometimes a programmer needs to pass a mutable object to a function but wants to make sure that the function does not modify the object at all. One method to avoid unwanted changes is to pass a copy of the object as the argument instead, like in the statement `my_func(num_list[:])`.

PARTICIPATION ACTIVITY 1.22.3: Modification of a list inside a function.

©zyBooks 12/15/22 00:10 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

zyDE 1.22.1: List argument modification.

Address the FIXME comments. Move the respective code from the while-loop to the (function. The add_grade function has already been created.

Note: split() and strip() are string methods further explained elsewhere. split() separa string into tokens using any whitespace as the default separator. The tokens are retu list (i.e., 'a b c'.split() returns ['a', 'b', 'c']). strip() returns a copy of a string with leading a trailing whitespace removed.

Load default ter

```
def add_grade(student_grades):
    print('Entering grade. \n')
    name, grade = input(grade_prompt).split()
    student_grades[name] = grade

# FIXME: Create delete_name function

# FIXME: Create print_grades function

student_grades = {}  # Create an empty dict
grade_prompt = "Enter name and grade (Ex. 'Bob A+'):\n"
delete_prompt = "Enter name to delete:\n"
menu_prompt = ("1. Add/modify student grade\n"
               "2. Delete student grade\n"
               "3. Print student grades\n"
               "4. Quit\n\n")

```

```
1
Johnny B+
1
```

Run

PARTICIPATION ACTIVITY 1.22.4: Arguments and mutability.

1) Assignments to a parameter name inside a function affect the code outside the function.

- ○ True
- ○ False

2) When a function is called, copies of all the argument objects are made.

- True
- False

3) Adding an element to a dictionary argument in a function might affect variables outside the function that reference the same dictionary object.

- True
- False

4) A programmer can protect mutable arguments from unwanted changes by passing a copy of the object to a function.

- True
- False

CHALLENGE ACTIVITY 1.22.1: Change order of elements in function list argument.

Write a function `swap` that swaps the first and last elements of a list argument.

Sample output with input: 'all,good,things,must,end,here'

```
['here', 'good', 'things', 'must', 'end', 'all']
```

422102.2723990.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
4 values_list = input().split(',')  # Program receives comma-separated values like 5,4,12,19
5 swap(values_list)
6
7 print(values_list)
```

Run

1.23 List slicing

A programmer can use ***slice notation*** to read multiple elements from a list, creating a new list that contains only the desired elements. The programmer indicates the start and end positions of a range of elements to retrieve, as in `my_list[0:2]`. The 0 is the position of the first element to read, and the 2 indicates last element. Every element between 0 and 2 from my_list will be in the new list. The end position, 2 in this case, is *not* included in the resulting list.

Figure 1.23.1: List slice notation.

```
boston_bruins = ['Tyler', 'Zdeno',
'Patrice']
print('Elements 0 and 1:',
boston_bruins[0:2])
print('Elements 1 and 2:',
boston_bruins[1:3])
```

```
Elements 0 and 1: ['Tyler',
'Zdeno']
Elements 1 and 2: ['Zdeno',
'Patrice']
```

The slice `boston_bruins[0:2]` produces a new list containing the elements in positions 0 and 1: ['Tyler', 'Zdeno']. The end position is *not* included in the produced list – to include the final element of a list in a slice, specify an end position past the end of the list. Ex: `boston_bruins[1:3]` produces the list ['Zdeno', 'Patrice'].

PARTICIPATION ACTIVITY 1.23.1: List slicing.

Animation captions:

1. The list object is created.
2. The list is sliced from 0 to 3, and then printed out.
3. The list is sliced from 1 up to 2.

Negative indices can also be used to count backwards from the end of the list.

Figure 1.23.2: List slicing: Using negative indices.

```
election_years = [1992, 1996, 2000, 2004, 2008]
print(election_years[0:-1])  # Every year except the last
print(election_years[0:-3])  # Every year except the last three
print(election_years[-3:-1])  # The third and second to last years
```

```
[1992, 1996, 2000, 2004]
[1992, 1996]
[2000, 2004]
```


A position of -1 refers to the last element of the list, thus election_years[0:-1] creates a slice containing all but the last election year. Such usage of negative indices is especially useful when the length of a list is not known, and is simpler than the equivalent expression election_years[0:len(election_years)-1].

PARTICIPATION ACTIVITY 1.23.2: List slicing.

Assume that the following code has been evaluated:

```
nums = [1, 1, 2, 3, 5, 8, 13]
```

1) What is the result of nums[1:5]?

Check Show answer

2) What is the result of nums[5:10]?

Check Show answer

3) What is the result of nums[3:-1]?

Check Show answer

An optional component of slice notation is the ***stride***, which indicates how many elements are skipped between extracted items in the source list. Ex: The expression my_list[0:5:2] has a stride of 2, thus skipping every other element, and resulting in a slice that contains the elements in positions 0, 2, and 4. The default stride value is 1 (the expressions my_list[0:5:1] and my_list[0:5] being equivalent).

If the reader has studied string slicing, then list slicing should be familiar. In fact, slicing has the same semantics for most sequence type objects.

PARTICIPATION ACTIVITY 1.23.3: List slicing.

Given the following code:

```
nums = [0, 25, 50, 75, 100]
```

1) The result of evaluating nums[0:5:2] is [25, 75].
   - True
   - False

2) The result of evaluating nums[0:-1:3] is [0, 75].
   - True
   - False

A table of common list slicing operations is given below. Note that omission of the start or end positions, such as my_list[:2] or my_list[4:], has the same meaning as in string slicing. my_list[:2] includes every element up to position 2. my_list[4:] includes every element following position 4 (including the element at position 4).

©zyBooks 12/15/22 00:10 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Table 1.23.1: Some common list slicing operations.

Operation	Description	Example code	Example output
my_list[start:end]	Get a list from start to end (minus 1).	`my_list = [5, 10, 20]` `print(my_list[0:2])`	`[5, 10]`
my_list[start:end:stride]	Get a list of every stride element from start to end (minus 1).	`my_list = [5, 10, 20, 40, 80]` `print(my_list[0:5:3])`	`[5, 40]`
my_list[start:]	Get a list from start to end of the list.	`my_list = [5, 10, 20, 40, 80]` `print(my_list[2:])`	`[20, 40, 80]`
my_list[:end]	Get a list from beginning of list to end (minus 1).	`my_list = [5, 10, 20, 40, 80]` `print(my_list[:4])`	`[5, 10, 20, 40]`
my_list[:]	Get a copy of the list.	`my_list = [5, 10, 20, 40, 80]` `print(my_list[:])`	`[5, 10, 20, 40, 80]`

©zyBooks 12/15/22 00:10 1361995 John Farrell COLOSTATECS220SeaboltFall2022

The interpreter handles incorrect or invalid start and end positions in slice notation gracefully. An end position that exceeds the length of the list is treated as the end of the list. If the end position is less than the start position, an empty list is produced.

PARTICIPATION ACTIVITY 1.23.4: Match the expressions to the list.

Match the expression on the left to the resulting list on the right. Assume that my_list is the following Fibonacci sequence:

```
my_list = [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

If unable to drag and drop, refresh the page.

my_list[2:5] **my_list[:20]** **my_list[4:]** **my_list[3:6]**

my_list[len(my_list)//2:(len(my_list)//2) + 1] **my_list[3:1]**

[5, 8, 13, 21, 34]

[]

[1, 1, 2, 3, 5, 8, 13, 21, 34]

[5]

[2, 3, 5]

[3, 5, 8]

Reset

CHALLENGE ACTIVITY 1.23.1: List slicing.

422102.2723990.qx3zqy7

1.24 Loops modifying lists

Sometimes a program iterates over a list while modifying the elements, such as by changing some elements' values, or by moving elements' positions.

Changing elements' values

The below example of changing element's values combines the len() and range() functions to

iterate over a list and increment each element of the list by 5.

Figure 1.24.1: Modifying a list during iteration example.

```
my_list = [3.2, 5.0, 16.5,
12.25]

for i in range(len(my_list)):
    my_list[ i ] += 5
```


The figure below shows two programs that each attempt to convert any negative numbers in a list to 0. The program on the right is incorrect, demonstrating a common logic error.

Figure 1.24.2: Modifying a list during iteration example: Converting negative values to 0.

Correct way to modify the list.

```
user_input = input('Enter
numbers: ')

tokens = user_input.split()

# Convert strings to
integers
nums = []
for token in tokens:
    nums.append(int(token))

# Print each position and
number
print()
for pos, val in
enumerate(nums):

    print(f'{pos}: {val}')

# Change negative values to
0
for pos in
range(len(nums)):
    if nums[pos] < 0:
        nums[pos] = 0

# Print new numbers
print('New numbers: ')
for num in nums:
    print(num, end=' ')
```

```
Enter numbers:5 67 -5 -4 5 6 6 4
0: 5
1: 67
2: -5
3: -4
4: 5
5: 6
6: 6
7: 4
New numbers:
5 67 0 0 5 6 6 4
```

Incorrect way: list not modified.


```
user_input = input('Enter
numbers:')

tokens = user_input.split()

# Convert strings to integers
nums = []
for token in tokens:
    nums.append(int(token))

# Print each position and number
print()
for pos, val in enumerate(nums):

    print(f'{pos}: {val}')

# Change negative values to 0
for num in nums:
    if num < 0:
        num = 0  # Logic error:
temp variable num set to 0

# Print new numbers
print('New numbers: ')
for num in nums:
    print(num, end=' ')
```

```
Enter numbers:5 67 -5 -4 5 6 6 4
0: 5
1: 67
2: -5
3: -4
4: 5
5: 6
6: 6
7: 4
New numbers:
5 67 -5 -4 5 6 6 4
```


The program on the right illustrates a common logic error. A common error when modifying a list during iteration is to update the loop variable instead of the list object. The statement `num = 0`

simply binds the name num to the integer literal value 0. The reference in the list is never changed.

In contrast, the program on the left correctly uses an index operation `nums[pos] = 0` to modify to 0 the reference held by the list in position pos. The below activities demonstrate further; note that only the second program changes the list's values.

PARTICIPATION ACTIVITY 1.24.1: Incorrect list modification example.

PARTICIPATION ACTIVITY 1.24.2: Corrected list modification example.

CHALLENGE ACTIVITY 1.24.1: Iterating through a list using range().

422102.2723990.qx3zqy7

PARTICIPATION ACTIVITY 1.24.3: List modification.

Consider the following program:

```
nums = [10, 20, 30, 40, 50]

for pos in range(len(nums)):
    tmp = nums[pos] / 2
    if (tmp % 2) == 0:
        nums[pos] = tmp
```

1) What's the final value of nums[1]?

Check Show answer

Changing list size

A common error is to add or remove a list element while iterating over that list. Such list modification can lead to unexpected behavior if the programmer is not careful. Ex: Consider the following program that reads in two sets of numbers and attempts to find numbers in the first set that are not in the second set.

Figure 1.24.3: Modifying lists while iterating: Incorrect program.

```
nums1 = []
nums2 = []

user_input = input('Enter first set of
numbers: ')
tokens = user_input.split()  # Split into
separate strings

# Convert strings to integers
print()
for pos, val in enumerate(tokens):
    nums1.append(int(val))

    print(f'{pos}: {val}')

user_input = input('Enter second set of
numbers:')
tokens = user_input.split()

# Convert strings to integers
print()
for pos, val in enumerate(tokens):
    nums2.append(int(val))

    print(f'{pos}: {val}')

# Remove elements from nums1 if also in
nums2
print()
for val in nums1:
    if val in nums2:

        print(f'Deleting {val}')
        nums1.remove(val)

# Print new numbers
print('\nNumbers only in first set:', end='
')
for num in nums1:
    print(num, end=' ')
```



```
Enter first set of
numbers:5 10 15 20
0: 5
1: 10
2: 15
3: 20
Enter second set of
numbers:15 20 25 30
0: 15
1: 20
2: 25
3: 30

Deleting 15

Numbers only in first set:
5 10 20
```


The above example iterates over the list nums1, deleting an element from the list if the element is also found in the list nums2. The programmer expected a certain result, namely that after removing an element from the list, the next iteration of the loop would reference the next element as normal. However, removing the element shifts the position of each following element in the list to the left by one. In the example above, removing 15 from nums1 shifts the value 20 left into position 2. The loop, having just iterated over position 2 and removing 15, moves to the next position and finds the

end of the list, thus never evaluating the final value 20.

The problem illustrated by the example above has a simple fix: Iterate over a copy of the list instead of the actual list being modified. Copying the list allows a programmer to modify, swap, add, or delete elements without affecting the loop iterations. The easiest way to copy the iterating list is to use slice notation inside of the loop expression, as in:

Figure 1.24.4: Copy a list using [:].


```
for item in
my_list[:]:
    # Loop
statements.
```

PARTICIPATION ACTIVITY 1.24.4: List modification.

Animation captions:

1. The loop, having just iterated over position 1 and removing 10, moves to the next position and finds the end of the list, thus never evaluating the final value 15.
2. The problem illustrated by the example above can be fixed by iterating over a copy of the list instead of the actual list being modified.

zyDE 1.24.1: Modify the above program to work correctly.

Modify the program (copied from above) using slice notation to iterate over a copy of

Load default template...

```
1
2  nums1 = []
3  nums2 = []
4
5  user_input = input('Enter first set of numb
6  tokens = user_input.split()  # Split into
7
8  # Convert strings to integers
9  for pos, val in enumerate(tokens):
10     nums1.append(int(val))
11
12     print(f'{pos}: {val}')
13
14 user_input = input('Enter second set of nur
15 tokens = user_input.split()
16
17 # Convert strings to integers
```

Pre-enter any input for program, th
run.

Run

PARTICIPATION ACTIVITY 1.24.5: Modifying a list while iterating.

1) Iterating over a list and deleting elements from the original list might cause a logic program error.
   - True
   - False

2) A programmer can iterate over a copy of a list to safely make changes to the original list.
   - True
   - False

1.25 Additional practice: Dice statistics

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

Analyzing dice rolls is a common example in understanding probability and statistics. The following program calculates the number of times the sum of two dice (randomly rolled) is equal to six or seven.

zyDE 1.25.1: Dice statistics.

Load default template...

```
1  import random
2
3  num_sixes = 0
4  num_sevens = 0
5  num_rolls = int(input('Enter number of roll
6
7  if num_rolls >= 1:
8      for i in range(num_rolls):
9          die1 = random.randint(1,6)
10         die2 = random.randint(1,6)
11         roll_total = die1 + die2
12
13         #Count number of sixes and sevens
14         if roll_total == 6:
15             num_sixes = num_sixes + 1
16         if roll_total == 7:
17             num_sevens = num_sevens + 1
```

6

Run

Create a different version of the program that:

1. Calculates the number of times the sum of the randomly rolled dice equals each possible value from 2 to 12.
2. Repeatedly asks the user for the number of times to roll the dice, quitting only when the user-entered number is less than 1. Hint: Use a while loop that will execute as long as num_rolls is greater than 1.
3. Prints a histogram in which the total number of times the dice rolls equals each possible value is displayed by printing a character, such as *, that number of times. The following provides an example:

```
Dice roll histogram:

2s:  **
3s:  ****
4s:  ***
5s:  ********
6s:  ************
7s:  ****************
8s:  ************
9s:  *********
10s: **********
11s: *****
12s: **
```


1.26 Additional practice: Number games

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

Several math games manipulate numbers in simple ways that yield fun results. Below is a program that takes any given 2-digit number and outputs a 6-digit number, having the 2-digits repeated. For example, 24 becomes 242424.

zyDE 1.26.1: Number game.

Enter a 2-digit number into the input box and press the run button.

Load default template...

```
1 num = int(input('Enter 2 digit number:\n'))
2
3 result = num * (3 * 7 * 13 * 37)
4
5 print(result)
6
```

24

Run

Create a different version of the program that:

1. Takes a 3-digit number and generates a 6-digit number with the 3-digit number repeated, for example, 391 becomes 391391. The rule is to multiply the 3-digit number by 7*11*13.
2. Takes a 5-digit number and generates a 10-digit number with the 5-digit number repeated, for example, 49522 becomes 4952249522. The rule is to multiply the 5-digit number by 11*9091.

Times 11: A two-digit number can be easily multiplied by 11 in one's head simply by adding the digits and inserting that sum between the digits. For example, 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 1.26.2: Number game.

Load default ter

```
# 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_ones) // 10)
#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, ',', num_in_ones)
```



```
5
8
```

Run

1.27 Additional practice: Health data

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following calculates a user's age in days based on the user's age in years.

zyDE 1.27.1: Health data: Age in days.

Load default template...

```
1 user_age_years = int(input('enter your age
2
3 user_age_days = user_age_years  * 365
4
5 print(f'You are at least {user_age_days} day
6
```

22

Run

Create a different version of the program that:

1. Calculates the user's age in minutes and seconds.
2. Estimates the approximate number of times the user's heart has beat in his/her lifetime using an average heart rate of 72 beats per minute.
3. Estimates the number of times the person has sneezed in his/her lifetime.
4. Estimates the number of calories that the person has expended in his/her lifetime (research on the Internet to obtain a daily estimate). Also calculate the number of sandwiches (or other common food item) that equals that number of calories.
5. Be creative: Pick several other interesting health-related statistics. Try searching the Internet to determine how to calculate that data, and create a program to perform that calculation. The program can ask the user to enter any information needed to perform the calculation.

1.28 LAB: List basics

This section's content is not available for print.

1.29 LAB: Set basics

This section's content is not available for print.

©zyBooks 12/15/22 00:10 1361995
John Farrell
COLOSTATECS220SeaboltFall2022