22.1 If-else branches (general)

Branch concept

People familiar with restaurants may be familiar with steering people to different-sized tables based on group size.

PARTICIPATION ACTIVITY 22.1.1: Branching concept.

Animation captions:

1. A restaurant host seats patrons. The host seats a party of 1 at the counter.
2. A party of 2 is seated at a small table. Other size parties are seated at a large table.
3. The host mentally executes the algorithm: If party of 1, seat at counter; Else If party of 2, seat at small table; Else seat at large table.

PARTICIPATION ACTIVITY 22.1.2: Branch concept.

Consider the example above.

1) A party of 1 is sat at _____ .
   - ○ the counter
   - ○ a small table

2) A party of 2 is sat at _____ .
   - ○ the counter
   - ○ a small table

3) A party of 5 is sat _____ .
   - ○ at a large table
   - ○ nowhere

Branch basics (If)

In a program, a ***branch*** is a sequence of statements only executed under a certain condition. Ex: A

hotel may discount a price only for people over age 65. An ***if*** branch is a branch taken only IF an expression is true.

PARTICIPATION ACTIVITY 22.1.3: Branches: Hotel rate example.

Animation content:


```
The animation executes the following Coral program:

integer hotelRate
integer userAge

hotelRate = 155
userAge = Get next input
if userAge > 65
   hotelRate = hotelRate - 20
Put "Your rate: " to output
Put hotelRate to output

Variables in memory is as follows:
135 hotelRate: integer
68 userAge: integer

Input is as follows:
68

Output (screen) is as follows:
Your rate: 135
```

Animation captions:

1. A decision leads to two program branches. If the expression is true, the first branch executes. Else, the second branch executes.
2. If userAge is 68, then 68 > 65 is true. So the first branch executes, which discounts hotelRate.
3. Execution rejoins the other branch, and continues with subsequent statements, outputting 135. If userAge were instead 50, the output would be 155.

PARTICIPATION ACTIVITY 22.1.4: Branches.

Consider the hotel rate example above.

1) If userAge is 20, does the true or false branch execute?
   - ○ True branch
   - ○ False branch

2) If userAge is 20, does the executed branch update hotelRate?
   - ○ Yes
   - ○ No

3) If userAge is 20, what hotel rate does the program output?
   - ○ 155
   - ○ 135

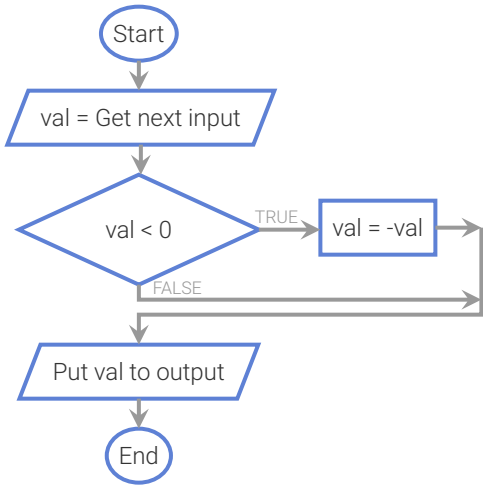
4) If userAge is 70, what hotel rate does the program output?
   - ○ 155
   - ○ 135

5) Do the last two statements always execute for any value of userAge?
   - ○ Yes
   - ○ No

If branch example: Absolute value

The example below shows how an if branch can be used to compute an absolute value of a number.

PARTICIPATION ACTIVITY 22.1.5: Computing absolute value. Full screen



Variables

0 val integer

Input

-99

Output

ENTER EXECUTION STEP RUN Execution speed Medium

PARTICIPATION ACTIVITY 22.1.6: Example if branch: Absolute value.

Consider the example above.

1) If the input is -6, does the branch execute?
   - Yes
   - No

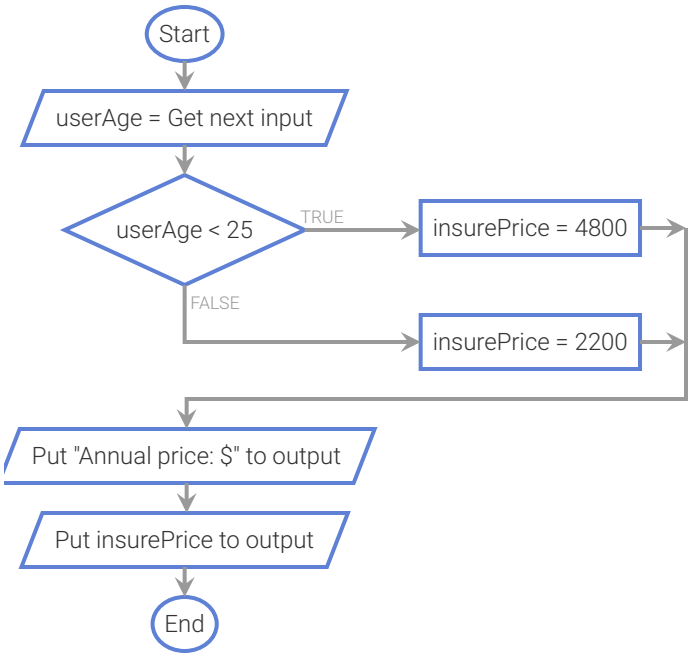
2) If the input is 0, does the branch execute?
   - Yes
   - No

If-else branches

An ***if-else*** branch has two branches: The first branch is executed IF an expression is true, ELSE the other branch is executed.

In the example below, if a user inputs an age less than 25, the statement `insurePrice = 4800` executes. Else, `insurePrice = 2200` executes.

PARTICIPATION ACTIVITY 22.1.7: Insurance price. Full screen



Variables

0	userAge	integer
0	insurePrice	integer

Input

22

Output

ENTER EXECUTION STEP RUN Execution speed Medium

Car insurance prices

(Car insurance prices for drivers under 25 are higher because 1 in 6 such drivers are involved in an accident each year, vs. 1 in 15 for older drivers. Source: www.census.gov, 2009).

PARTICIPATION ACTIVITY 22.1.8: If-else branches.

Consider the insurance price example above.

1) If userAge is 18, what price is output?

- 4800
- 2200

2) If userAge is 30, what price is output?

- 4800
- 2200

3) If userAge is 25, what price is output?

- 4800
- 2200

4) For what value of userAge will both branches execute?

- 15
- 25
- None

5) For what value of userAge will neither branch execute?

- 30
- 25
- None

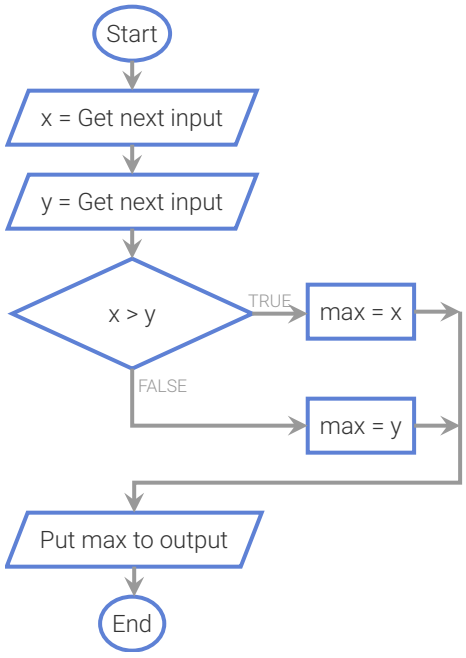
6) For what value of userAge will the output statements not execute?

- 20
- 25
- None

If-else example: Max

The example below shows how an if-else can be used to get the maximum of two values.

PARTICIPATION ACTIVITY 22.1.9: If-else branches example: Max. Full screen



Variables

0	x	integer
0	y	integer
0	max	integer

Input

55 79

Output

ENTER EXECUTION STEP RUN Execution speed Medium

PARTICIPATION ACTIVITY 22.1.10: If-else example: Max.

Consider the example above.

1) When the input is -3 0, which branch executes?
   - If
   - Else

2) When the input is 99 98, which branch executes?
   - If
   - Else

3) The if branch assigns max = x. The else branch assigns max = ?
   - x
   - y

4) If the inputs are 5 5, does max get assigned with x or y?

- x
- y

If-elseif-else branches

Commonly a programmer wishes to take one of multiple (three or more) branches. An if-else can be extended to an if-elseif-else structure. Each branch's expression is checked in sequence, as soon as one branch's expression is found to be true, that branch is taken. If no expression is found true, execution will reach the else branch, which then executes.

Note: The else part is optional. If omitted, then if none of the previous expressions are true, no branch executes.

PARTICIPATION ACTIVITY 22.1.11: If-elseif example: Anniversaries.

Animation content:

The animation executes the following Coral program:

```
integer numYears

numYears = Get next input

if numYears == 1
  Put "Newlyweds" to output
else
  if numYears == 25
     Put "Silver" to output
  else
     if numYears == 50
        Put "Golden" to output
     else
        Put "Congrats" to output
```



```
Variables in memory is as follows:
0 numYears: integer
```

Animation captions:

1. This program detects the specific value of a variable. If numYears is 1, the first branch executes and "Newlyweds" is output.
2. Else, if numYears is 25, the second branch executes and "Silver" is output. Else, if numYears is 50, the third branch executes and "Golden" is output.
3. Else, the last branch executes.

PARTICIPATION ACTIVITY 22.1.12: If-elseif-else.

Consider the if-elseif-else structure below:

```
if x equals -1
     Put "Disagrees" to output
else if x equals 0
     Put "Neutral" to output
else if x equals 1
     Put "Agrees" to output
else
     Put "Invalid entry" to output
```

1) If x is 1, what is output?

- Disagrees
- Neutral
- Agrees
- Invalid entry

2) If x is -2, what is output?

- Disagrees
- Invalid entry
- (Nothing is output)

3) Could the programmer have written the three branches in the order x equals 1, x equals 0, and x equals -1, and achieved the same results?

- No
- Yes

4) In the code above, suppose a programmer, after the third branch (x equals 1), inserts a new branch: Else If x equals -1 ... When might that new branch execute?

- When x is -1
- When x is 1
- Never

5) In the code above, suppose a programmer removed the Else part entirely. If x is 2, which is correct?
   - The last branch, meaning the Else If x equals 1 branch, will execute.
   - No branch will execute.
   - The program is not legal.

CHALLENGE ACTIVITY 22.1.1: If-else branches.

422102.2723990.qx3zqy7

22.2 Membership and identity operators

Membership operators: in/not in

One common programming task involves determining whether a specific value can be found within a container, such as a list or dictionary. The ***in*** and ***not in*** operators, known as ***membership operators***, yield True or False if the left operand matches the value of some element in the right operand, which is always a container.

PARTICIPATION ACTIVITY 22.2.1: Membership operators: Checking for a value in a list.

Animation content:

undefined

Animation captions:

1. A user creates a new list.
2. Every element in the list is checked for the value of 15.

3. Every element in the list is checked for the value of 44, which is not found. So the statement (44 in prices) evaluates to False.

The membership operators can be used with sequence types. If the variable x is a list or tuple, then `a in x` evaluates to True if there exists an index idx for which `a == x[idx]` is True. The program below demonstrates membership operator usage in a list:

Figure 22.2.1: Membership operators example: Checking for an item in a list.

```
# Use the "in" operator
barcelona_fc_roster = ['Alves',
'Messi', 'Fabregas']

name = input('Enter name to
check: ')

if name in barcelona_fc_roster:
    print('Found', name, 'on
the roster.')
else:
    print('Could not find',
name, 'on the roster.')
```

```
Enter name to check: Messi
Found Messi on the roster.
...
Enter name to check: Rooney
Could not find Rooney on the roster.
```

```
# Use the "not in" operator
barcelona_fc_roster = ['Alves',
'Messi', 'Fabregas']

name = input('Enter name to
check: ')

if name not in
barcelona_fc_roster:
    print('Could not find',
name, 'on the roster.')
else:
    print('Found', name, 'on
the roster.')
```

```
Enter name to check: Messi
Found Messi on the roster.
...
Enter name to check: Rooney
Could not find Rooney on the roster.
```

Membership operators can be used to check whether a string is a ***substring***, or matching subset of characters, of a larger string. For example, `'abc' in '123abcd'` returns True because the substring abc exists in the larger string.

Figure 22.2.2: Checking for substrings.

```
request_str = 'GET index.html
HTTP/1.1'

if '/1.1' in request_str:
    print('HTTP protocol 1.1')

if 'HTTPS' not in request_str:
    print('Unsecured connection')
```

```
HTTP protocol 1.1
Unsecured
connection
```


Membership in a dictionary implies that a specific *key* exists in the dictionary. A common error is to assume that a membership operator checks the values of each dictionary key as well.

Figure 22.2.3: Checking for membership in a dict.

```
my_dict = {'A': 1, 'B': 2, 'C': 3}

if 'B' in my_dict:
   print("Found 'B'")
else:
   print("'B' not found")

# Membership operator does not check
values
if 3 in my_dict:
    print('Found 3')
else:
    print('3 not found')
```

```
Found 'B'
3 not
found
```

PARTICIPATION ACTIVITY 22.2.2: Membership operators.

1) Which expression checks whether the list my_list contains the value 15?

- 15 in my_list[0]
- 15 in my_list
- my_list['15'] != 0

2) Which expression checks if the value 10 exists in the dictionary my_dict?

- 10 in my_dict['key']
- 10 in my_dict
- None of the above

Identity operators: is/is not

Sometimes a programmer wants to determine whether two variables are the same object. The programmer can use the ***identity operator***, ***is***, to check whether two operands are bound to a single object. The inverse identity operator, ***is not***, gives the negated value of 'is'. Thus, if `x is y` is True, then `x is not y` is False.

Identity operators do not compare object values; rather, identity operators compare object identities to determine equivalence. Object identity is usually [1] the memory address of an object. Thus, identity operators return True only if the operands reference the same object.

A common error is to confuse the equivalence operator "==" and the identity operator "is", because a statement such as `if x is 3` is valid syntax and is grammatically appealing. Python may confusedly evaluate the statement `x is 3` as True, but `y is 1000` as False, when x = 3 and y = 1000. Python interpreters typically precreate objects for a small range of numbers to avoid constantly recreating objects for such small values. In the example above, an object for 3 was precreated and thus x references the same object as the literal. However, Python did not precreate an object for 1000. A good practice is to avoid using the identity operators "is" and "is not", unless explicitly testing whether two objects are identical.

The id() function can be used to retrieve the identifier of any object. If `x is y` is True, then `id(x) == id(y)` is also True.

Figure 22.2.4: Identity operators.

```
w = 500
x = 500 + 500  # Create a new object with
value 1000
y = w + w      # Create a second object
with value 1000
z = x          # Bind z to the same
object as x

if z is x:
    print('z and x are bound to the same
object')
if z is not y:
    print('z and y are NOT bound to the
same object')
```

```
z and x are bound to the same
object
z and y are NOT bound to the
same object
```


PARTICIPATION ACTIVITY 22.2.3: Membership and identity operators.

Write the simplest expression that captures the desired comparison.

1) x is a key in the dict my_dict

Check Show answer

2) The variables x and y are unique objects.

Check Show answer

3) The character 'G' exists in the string my_str

Check Show answer

4) my_str is not the third element

in the list my_list

Check Show answer

CHALLENGE ACTIVITY 22.2.1: Membership and Identity: Enter the output of the code.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 22.2.2: Boolean operators: Detect specific values.

Write an expression using membership operators that prints "Special number" if special_num is one of the special numbers stored in the list special_list = [-99, 0, 44].

Sample output with input: 17

```
Not special number
```

422102.2723990.qx3zqy7

```
special_list = [-99, 0, 44]
special_num = int(input())

if ''' Your solution goes here ''':
    print('Special number')
else:
    print('Not special number')
```


Run

(*1) Object identity is an implementation detail of Python. For the standard CPython

implementation, identity is the memory address of the object.

22.3 Order of evaluation

Precedence rules

The order in which operators are evaluated in an expression is known as ***precedence rules***. Arithmetic, logical, and relational operators are evaluated in the order shown below.

©zyBooks 12/15/22 00:43 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Table 22.3.1: Precedence rules for arithmetic, logical, and relational operators.

Operator/Convention	Description	Explanation
( )	Items within parentheses are evaluated first	In `(a * (b + c)) - d`, the + is evaluated first, then *, then -.
* / % + -	Arithmetic operators (using their precedence rules; see earlier section)	`z - 45 * y < 53` evaluates * first, then -, then <.
< <= > >= == !=	Relational, (in)equality, and membership operators	`x < 2 or x >= 10` is evaluated as `(x < 2) or (x >= 10)` because < and >= have precedence over or.
not	not (logical NOT)	`not x or y` is evaluated as `(not x) or y`
and	Logical AND	`x == 5 or y == 10 and z != 10` is evaluated as `(x == 5) or ((y == 10) and (z != 10))` because and has precedence over or.
or	Logical OR	`x == 7 or x < 2` is evaluated as `(x == 7) or (x < 2)` because < and == have precedence over or

PARTICIPATION ACTIVITY 22.3.1: Applying the precedence rules to an expression can be thought of as a 'tree'.

Animation content:

undefined

Animation captions:

1. Expressions like x + 1 > y * z or z == 3 are evaluated using precedence rules. Among +, >, *, or, and ==, the * comes first.
2. Next comes +, >, ==, and finally or.
3. The expression is actually treated like a "tree", evaluated from the bottom upwards.
4. If x is 7, y is 6, and z is 3, then y * z is 18. Next, x + 1 is 8. Next, 8 > 18 is False. Next, z == 3 is True. Finally, False or True is True.

PARTICIPATION ACTIVITY 22.3.2: Order of evaluation.

To teach precedence rules, these questions intentionally omit parentheses; good style would use parentheses to make order of evaluation explicit.

1) Which operator is evaluated first?

```
not y and x
```

- and
- not

2) Which operator has precedence?

```
w + 3 > x - y * z
```

- +
- -
- >
- *

3) In what order are the operators evaluated?

```
w + 3 != y - 1 and x
```

- +, !=, -, and
- +, -, and, !=
- +, -, !=, and

4) To what does this expression evaluate, given `x = 4`, `y = 7`.

`x == 3 or x + 1 > y`

- True
- False

Common error: Missing parentheses

A common error is to write an expression that is evaluated in a different order than expected. Good practice is to use parentheses in expressions to make the intended order of evaluation explicit. For example, a programmer might write:

- `not a == b` intending to mean `(not a) == b`, but in fact the interpreter computes `not (a == b)` because equality operators (==) have precedence over logical operations (not).
- `w and x == y and z` intending `(w and x) == (y and z)`, but the interpreter computes `(w and (x == y)) and z` because == has precedence over and.
- `not x + y < 5` intending `(not x) + y < 5`, but the interpreter computes `not ((x + y) < 5)` because the addition operator + has the highest precedence and is computed first, followed by the relational operation <, and finally the logical not operation.

PARTICIPATION ACTIVITY 22.3.3: Common errors in expressions.

1) `not x == 3` evaluates as `not (x == 3)`.
   - Yes
   - No

2) `w + x == y + z` evaluates as `(w + x) == (y + z)`.
   - Yes
   - No

3) `w and x == y and z` evaluates as `(w and x) == (y and z)`.
   - Yes
   - No

PARTICIPATION ACTIVITY 22.3.4: Order of evaluation.

Which illustrates the actual order of evaluation via parentheses?

1) `not green == red`

- `(not green) == red`
- `not (green == red)`
- `(not green =)= red`

2) `bats < birds or birds < insects`

- `((bats < birds) or birds) < insects`
- `bats < (birds or birds) < insects`
- `(bats < birds) or (birds < insects)`

3) `not (bats < birds) or (birds < insects)`

- `not ((bats < birds) or (birds < insects))`
- `(not (bats < birds)) or (birds < insects)`
- `((not bats) < birds) or (birds < insects)`

4) `(num1 == 9) or (num2 == 0) and (num3 == 0)`

- `(num1 == 9) or ((num2 == 0) and (num3 == 0))`
- `((num1 == 9) or (num2 == 0)) and (num3 == 0)`
- `(num1 == 9) or (num2 == (0 and num3) == 0)`

22.4 Code blocks and indentation

Code blocks

A ***code block*** is a series of statements that are grouped together. A code block in Python is defined by its indentation level, i.e., the number of blank columns from the left edge. The initial code block is not indented. A new code block can follow a statement that ends with a colon, such as an "if" or "else". In addition, a new code block must be more indented than the previous code block. The program below includes comments indicating where each new code block begins.

The amount of indentation used to indicate a new code block can be arbitrary, as long as the programmer uses the same indentation consistently for each line in the block. Good practice is to use the standard recommended 4 columns per indentation level.

A common error for new Python programmers is the mixing of tabs and spaces. Never mix tabs and spaces for indentation in the same program. Many editors consider a tab to be equivalent to either 3 or 4 spaces, while in Python a tab is equivalent only to another tab. A program that mixes tabs and spaces to indent code blocks will automatically generate an IndentationError from the interpreter in Python 3. A good practice is to use spaces only when indenting code, and to set text editor options to automatically use spaces when possible.

©zyBooks 12/15/22 00:43 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

Figure 22.4.1: Code blocks are indicated with indentation.

```
# First code block has no indentation

model = input('Enter car model: ')
year = int(input('Enter year of car
manufacture: '))

antique = False
domestic = False

if year < 1970:
    # New code block has indentation of 4
columns
    antique = True

# Back to code block 0

if model in ['Ford', 'Chevrolet',
'Dodge']:
  # New code block has indentation of 2
columns
  # Any amount of indentation > 0 is OK.
  domestic = True

# Back to code block 0

if antique:
    # New code block has indentation of 4
columns
    if domestic:
        # New block has 4 additional
columns (8 total)
        print('My own model-T still runs
like a charm...')
```

```
Enter car model: Ford
Enter year of car manufacture:
1918
My own model-T still runs like
a charm...
```


©zyBooks 12/15/22 00:43 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

zyDE 22.4.1: Code blocks and indentation.

Fix the errors in the code below so that you can run the program.

Load default ter

```
# Calculate the velocity required to escape a circular orbit of Earth,
# depending on a user-entered orbital distance.

 import math

escape_velocity_meters_per_sec = 0  # Required velocity to escape orbit
grav_constant = 6.67384e-11   # Earth's gravitational constant
earth_mass_kilograms = 5.972e24      # Mass of the Earth

    radius_meters = float(input('Enter distance from center of Earth: '))
    print()

if radius_meters < 6317000:  # 6317 km is the average radius of the Earth
    escape_velocity_meters_per_sec = 0  # No escape possible!
        print('Houston, we have a problem')


```


160000

Run

Special cases

The number of columns of text considered to be acceptable varies from 80 to 120. Good practice is to use the widely accepted standard of 80 columns. A few exceptions to the rules of indentation deal with very long statements that require more than one line and *wrap* to the next line. Such special situations do not indicate new code blocks.

Figure 22.4.2: Some indentations are continuations of the previous line.

Multiple lines enclosed within parentheses are implicitly joined into a single string (without newlines between each line); use implicit line joining for very long strings or functions with numerous arguments. Ex: All extra lines are indented to the same column as the opening quotation mark on the first line.

When declaring list or dict literals, entries can be placed on separate lines for clarity.

```
declaration = ("When in the Course of human events, it becomes necessary
for "
               "one people to dissolve the political bands which have
connected "
               "them with another, and to assume among the powers of the
earth...")

result_of_power = math.pow(long_variable_name_left_operand,
                           long_variable_name_right_operand)
```

Figure 22.4.3: List, dict multi-line constructs.

Containers like lists and dicts can be broken into multiple lines, with the elements on separate, indented lines.

```
my_list = [
    1, 2, 3,
    4, 5, 6
    ]
```

```
my_dict = {
    'entryA': 1,
    'entryB': 2
}
```

PARTICIPATION ACTIVITY 22.4.1: Indentation.

1) The standard number of spaces to use for indentation is 4.

   - True
   - False

2) Mixing spaces and tabs when indenting is considered an acceptable

programming style.

- True
- False

3) A programmer can start new code blocks at any point in the code, as long as the indentation for each line in the block is consistent.

- True
- False

CHALLENGE ACTIVITY 22.4.1: Indentation: Fix the program.

Retype the below code. Fix the indentation as necessary to make the program work.

```
    if 'New York' in temperatures:
        if temperatures['New York'] > 90:
        print('The city is melting!')
        else:
            print('The temperature in New York is',
temperatures['New York'])
else:
        print('The temperature in New York is unknown.')
```

Sample output with input: 105

```
The city is melting!
```

422102.2723990.qx3zqy7

```
1 temperatures = {
2     'Seattle': 56.5,
3     'New York': float(input()),
4     'Kansas City': 81.9,
5     'Los Angeles': 76.5
6 }
7
8 ''' Your solution goes here '''
9
```

Run

22.5 Conditional expressions

Conditional expressions

A ***conditional expression*** has the following form:

Construct 22.5.1: Conditional expression.

```
expr_when_true if condition else
expr_when_false
```

All three operands are expressions. The condition in the middle is evaluated first. If the condition evaluates to True, then expr_when_true is evaluated. If the condition evaluates to False, then expr_when_false is evaluated. For example, if x is 2, then the conditional expression `5 if x==2 else 9*x` evaluates to 5.

A conditional expression has three operands and thus is sometimes referred to as a ***ternary operation***.

Good practice is to restrict usage of conditional expressions to an assignment statement, as in: `y = 5 if (x == 2) else 9*x`. Some Python programmers denounce conditional expressions as difficult to read and comprehend, since the middle operand is actually the first evaluated, and left-to-right syntax is preferred. However, simple assignments such as the statement above are acceptable.

PARTICIPATION ACTIVITY 22.5.1: Conditional expression.

Animation captions:

1. This if-else form can be written as a conditional expression.
2. The condition in the middle is evaluated first.
3. If the condition evaluates to True, then expr1 is evaluated and my_var is assigned with

expr1's value.

4. If the condition evaluates to False, then expr2 is evaluated and my_var is assigned with expr2's value.

PARTICIPATION ACTIVITY 22.5.2: Conditional expressions.

Convert each if-else statement to a single assignment statement using a conditional expression, using parentheses around the condition. Enter "Not possible" if appropriate.

1)
```
if x < 100:
    y = 0
else:
    y = x
```

Check Show answer

2)
```
if x < 0:
    x = -x
else:
    x = x
```

Check Show answer

3)
```
if x < 1:
    y = x
else:
    z = x
```

Check Show answer

CHALLENGE ACTIVITY 22.5.1: Conditional expressions: Enter the output of the code.

422102.2723990.qx3zqy7

CHALLENGE ACTIVITY 22.5.2: Conditional expression: Print negative or non-negative.

Create a conditional expression that evaluates to string "negative" if user_val is less than 0, and "non-negative" otherwise.

Sample output with input: -9

```
-9 is negative
```


422102.2723990.qx3zqy7

```
1 user_val = int(input())
2
3 cond_str = ''' Your solution goes here '''
4
5 print(user_val, 'is', cond_str)
```

Run

CHALLENGE ACTIVITY 22.5.3: Conditional expression: Conditional assignment.

Using a conditional expression, write a statement that increments num_users if update_direction is 3, otherwise decrements num_users.

Sample output with inputs: 8 3

```
New value is: 9
```


422102.2723990.qx3zqy7

```
1 num_users = int(input())
2 update_direction = int(input())
3
```

Run

22.6 Additional practice: Tweet decoder

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 program decodes a few common abbreviations in online communication such as messages in Twitter ("tweets") or email, and provides the corresponding English phrase.

©zyBooks 12/15/22 00:43 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

zyDE 22.6.1: Tweet decoder.

Load default template...

```
tweet = input('Enter abbreviation from twee

if tweet == 'LOL':
    print('LOL = laughing out loud')
elif tweet == 'BFN':
    print('BFN = bye for now')
elif tweet == 'FTW':
    print('FTW = for the win')
elif tweet == 'IRL':
    print('IRL = in real life')
else:
    print("Sorry, don't know that one")

```

LOL

Run

Create different versions of the program that:

1. Expand the number of abbreviations that can be decoded. Add support for abbreviations you commonly use or search the internet for some.
2. Allow the user to enter a complete tweet (160 characters or less) as a single line of text. Search the resulting string for abbreviations and print a list of each abbreviation along with its decoded meaning.

22.7 LAB: Remove gray from RGB

Summary: Given integer values for red, green, and blue, subtract the gray from each value.

Computers represent color by combining the sub-colors red, green, and blue (rgb). Each sub-color's value can range from 0 to 255. Thus (255, 0, 0) is bright red, (130, 0, 130) is a medium purple, (0, 0, 0) is black, (255, 255, 255) is white, and (40, 40, 40) is a dark gray. (130, 50, 130) is a faded purple, due to the (50, 50, 50) gray part. (In other words, equal amounts of red, green, blue yield gray).

Given values for red, green, and blue, remove the gray part.

Ex: If the input is:

```
130
50
```

```
130
```

the output is:

```
80 0 80
```

Find the smallest value, and then subtract it from all three values, thus removing the gray.

Note: This page converts rgb values into colors.

422102.2723990.qx3zqy7

LAB ACTIVITY | 22.7.1: LAB: Remove gray from RGB | 0 / 10

main.py — Load default template...

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

Develop mode | **Submit mode**

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

Enter program input (optional)

If your code requires input values, provide them here.

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```


22.8 LAB: Smallest number

Write a program whose inputs are three integers, and whose output is the smallest of the three values.

Ex: If the input is:

```
7
15
3
```

the output is:

```
3
```

422102.2723990.qx3zqy7

LAB ACTIVITY | 22.8.1: LAB: Smallest number | 0 / 10

main.py Load default template...

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

Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

Run program

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

22.9 LAB: Interstate highway numbers

Primary U.S. interstate highways are numbered 1-99. Odd numbers (like the 5 or 95) go north/south, and evens (like the 10 or 90) go east/west. Auxiliary highways are numbered 100-999, and service the primary highway indicated by the rightmost two digits. Thus, I-405 services I-5, and I-290 services I-90. Note: 200 is not a valid auxiliary highway because 00 is not a valid primary highway number.

Given a highway number, indicate whether it is a primary or auxiliary highway. If auxiliary, indicate what primary highway it serves. Also indicate if the (primary) highway runs north/south or east/west.

Ex: If the input is:

```
90
```

the output is:

```
I-90 is primary, going east/west.
```

Ex: If the input is:

```
290
```


the output is:

```
I-290 is auxiliary, serving I-90, going east/west.
```

Ex: If the input is:

```
0
```

the output is:

```
0 is not a valid interstate highway number.
```

Ex: If the input is:

```
200
```

the output is:

```
200 is not a valid interstate highway number.
```

See Wikipedia for more info on highway numbering.

422102.2723990.qx3zqy7

LAB ACTIVITY 22.9.1: LAB: Interstate highway numbers 0 / 10

main.py Load default template...


```
highway_number = int(input())

''' Type your code here. '''

```

Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

22.10 LAB: Seasons

Write a program that takes a date as input and outputs the date's season in the northern hemisphere. The input is a string to represent the month and an int to represent the day.

Ex: If the input is:

```
April
11
```

the output is:

```
Spring
```

In addition, check if the string and int are valid (an actual month and day).

Ex: If the input is:

```
Blue
65
```


the output is:

```
Invalid
```

The dates for each season in the northern hemisphere are:
Spring: March 20 - June 20
Summer: June 21 - September 21
Autumn: September 22 - December 20
Winter: December 21 - March 19

422102.2723990.qx3zqy7

LAB ACTIVITY | 22.10.1: LAB: Seasons | 0 / 10

main.py — Load default template...

```
1 input_month = input()
2 input_day = int(input())
3
4 ''' Type your code here. '''
5
```


Develop mode | **Submit mode**

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

Enter program input (optional)

If your code requires input values, provide them here.

Run program

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

22.11 LAB: Exact change

Write a program with total change amount as an integer input, and output the change using the fewest coins, one coin type per line. The coin types are Dollars, Quarters, Dimes, Nickels, and Pennies. Use singular and plural coin names as appropriate, like 1 Penny vs. 2 Pennies.

Ex: If the input is:

```
0
```

(or less than 0), the output is:

```
No change
```

Ex: If the input is:


```
45
```

the output is:

```
1 Quarter
2 Dimes
```

422102.2723990.qx3zqy7

LAB ACTIVITY | 22.11.1: LAB: Exact change | 0 / 10

main.py Load default template...

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


Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

Run program

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

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

22.12 LAB: Leap year

A year in the modern Gregorian Calendar consists of 365 days. In reality, the earth takes longer to rotate around the sun. To account for the difference in time, every 4 years, a leap year takes place. A leap year is when a year has 366 days: An extra day, February 29th. The requirements for a given year to be a leap year are:

1) The year must be divisible by 4

2) If the year is a century year (1700, 1800, etc.), the year must be evenly divisible by 400; therefore, both 1700 and 1800 are not leap years

Some example leap years are 1600, 1712, and 2016.

Write a program that takes in a year and determines whether that year is a leap year.

Ex: If the input is:

```
1712
```

the output is:

```
1712 - leap year
```

Ex: If the input is:

```
1913
```

the output is:

```
1913 - not a leap year
```

422102.2723990.qx3zqy7

LAB ACTIVITY | 22.12.1: LAB: Leap year | 0 / 10

main.py Load default template...

```
1 is_leap_year = False
2
3 input_year = int(input())
4
5 ''' Type your code here. '''
```

Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

Run program

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

22.13 LAB: Warm up: Automobile service cost

(1) Prompt the user for an automobile service. Output the user's input. (1 pt)

Ex:

```
Enter desired auto service:
Oil change
You entered: Oil change
```

(2) Output the price of the requested service. (4 pts)

Ex:

```
Enter desired auto service:
Oil change
You entered: Oil change
Cost of oil change: $35
```


The program should support the following services (all integers):

- Oil change -- $35
- Tire rotation -- $19
- Car wash -- $7

If the user enters a service that is not listed above, then output the following error message:

```
Error: Requested service is not recognized
```

422102.2723990.qx3zqy7

LAB ACTIVITY | 22.13.1: LAB: Warm up: Automobile service cost | 0 / 5

main.py Load default template...

```
1 # Type your code here
```

Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

Run program

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

22.14 LAB*: Program: Automobile service invoice

(1) Output a menu of automotive services and the corresponding cost of each service. (2 pts)

Ex:

```
Davy's auto shop services
Oil change -- $35
Tire rotation -- $19
Car wash -- $7
Car wax -- $12
```


(2) Prompt the user for two services from the menu. (2 pts)

Ex:

```
Select first service:
Oil change
Select second service:
Car wax
```


(3) Output an invoice for the services selected. Output the cost for each service and the total cost. (3 pts)

```
Davy's auto shop invoice

Service 1: Oil change, $35
Service 2: Car wax, $12

Total: $47
```

(4) Extend the program to allow the user to enter a dash (-), which indicates no service. (3 pts)

Ex:

```
Davy's auto shop services
Oil change -- $35
Tire rotation -- $19
Car wash -- $7
Car wax -- $12

Select first service:
Tire rotation
Select second service:
-

Davy's auto shop invoice

Service 1: Tire rotation, $19
Service 2: No service
```

```
Total: $19
```

422102.2723990.qx3zqy7

LAB ACTIVITY 22.14.1: LAB*: Program: Automobile service invoice 0 / 10

main.py Load default template...

```
1 # Type your code here
```

Develop mode | **Submit mode**

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

Enter program input (optional)

If your code requires input values, provide them here.

Run program

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

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

22.15 LAB: Golf scores

This section's content is not available for print.

©zyBooks 12/15/22 00:43 1361995
John Farrell
COLOSTATECS220SeaboltFall2022