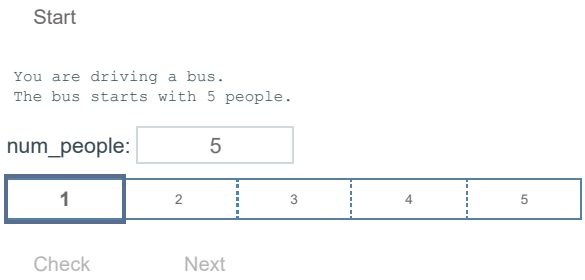
2.1 Variables and assignments

Remembering a value

Here's a variation on a common school child riddle.

CHALLENGE ACTIVITY 2.1.1: People on bus.

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



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

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

Variables and assignments

In a program, a ***variable*** is a named item, such as x or num_people, used to hold a value.

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

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

PARTICIPATION ACTIVITY 2.1.1: Variables and assignments.

Animation captions:

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

= is not equals

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

PARTICIPATION ACTIVITY 2.1.2: Valid assignment statements.

Indicate which assignment statements are valid.

1) x = 1
   - ○ Valid
   - ○ Invalid

2) x = y
   - ○ Valid
   - ○ Invalid

3) x = y + 2
   - ○ Valid
   - ○ Invalid

4) x + 1 = 3
   - ○ Valid
   - ○ Invalid

5) x + y = y + x
   - ○ Valid
   - ○ Invalid

PARTICIPATION ACTIVITY 2.1.3: Variables and assignment statements.

Given variables x, y, and z.

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

   Check **Show answer**

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

   Check **Show answer**

3) x = 9
   y = x + 1

x = 5
What is y?

Check Show answer

PARTICIPATION ACTIVITY 2.1.4: Trace the variable value.

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

Start

```
x = 7
y = 7
z = 2
x = 1
y = 4
z = 1
x = 6
```

x is
1 6 7

y is
4 0 7

z is
6 2 1

1 2 3 4

Check Next

Assignments with variable on left and right

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

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

PARTICIPATION ACTIVITY 2.1.5: A variable may appear on the left and right of an assignment statement.

Animation captions:

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

PARTICIPATION ACTIVITY 2.1.6: Variable on both sides.

Indicate the value of x after the statements execute.

1) x = 5
x = x + 7

Check Show answer

2)

```
x = 2
y = 3

x = x * y
x = x * y
```

Check Show answer

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

Check Show answer

4) Complete this statement to increment y: y = _____

Check Show answer

CHALLENGE ACTIVITY 2.1.2: Assigning a sum.

Write a statement that assigns total_coins with the sum of nickel_count and dime_count.
Sample output for 100 nickels and 200 dimes is:

```
300
```

```
total_coins = 0

nickel_count = int(input())
dime_count = int(input())

''' Your solution goes here '''

print(total_coins)
```

Run

View solution *(Instructors only)*

Download student submissions

CHALLENGE ACTIVITY 2.1.3: Multiplying the current value of a variable.

Write a statement that assigns cell_count with cell_count multiplied by 10. * performs multiplication. If the input is 10, the output should be:

```
100
```

```
1 cell_count = int(input())
2
3 ''' Your solution goes here '''
4
5 print(cell_count)
```


Run

View solution *(Instructors only)*

Download student submissions

2.2 Identifiers

Rules for identifiers

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

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

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

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

PARTICIPATION ACTIVITY 2.2.1: Valid names.

Which of the following are valid names?

1) numCars

- ○ Valid
- ○ Invalid

2) num_cars1

- Valid
- Invalid

3) _num_cars

- Valid
- Invalid

4) ___numcars2

- Valid
- Invalid

5) num cars

- Valid
- Invalid

6) 3rd_place

- Valid
- Invalid

7) third_place_

- Valid
- Invalid

8) third_place!

- Valid
- Invalid

9) output

- Valid
- Invalid

10) print_copy

- Valid
- Invalid

Style guidelines for identifiers

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

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

Table 2.2.1: Use meaningful variable names.

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

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

Table 2.2.2: Python 3 reserved keywords.

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

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

PARTICIPATION ACTIVITY 2.2.2: Python 3 name validator.

Use the tool below to test valid and invalid names.

Try an identifier: Validate

Awaiting your input...

2.3 Objects

Objects

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

interpreter creates and manipulates objects as needed to run the Python code. Objects are used to represent everything in a Python program, including integers, strings, functions, lists, etc.

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

PARTICIPATION ACTIVITY 2.3.1: Creating new objects.

Animation captions:

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

Above, the interpreter performs an addition of 2+2, resulting in a new object being created with a value of 4. Once 4 is printed the object is no longer needed, so the object is automatically deleted from memory and thrown away. Deleting unused objects is an automatic process called ***garbage collection*** that helps to keep the memory of the computer less utilized.

Name binding

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

PARTICIPATION ACTIVITY 2.3.2: Manipulating variables.

Animation captions:

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

Properties of objects

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

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

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

Figure 2.3.1: Printing displays an object's value.


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

```
4
5
```

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

The type of an object also determines the mutability of an object. ***Mutability*** indicates whether the object's value is allowed to be changed. Integers and strings are ***immutable***; modifying their values with assignment statements results in new objects being created and the names bound to the new object.

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


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

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

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

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

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

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

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

```
1752608
2330312
```

zyDE 2.3.1: Experimenting with objects.

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

Load default template... **Run**

```
1  birthday_year = 1986
2  birthday_month = 'April'
3  birthday_day = 22
4
5  print('birthday_year -->')
6  print(' value:', str(birthday_year))
7  print(' type:', type(birthday_year))
8  print(' id:', id(birthday_year))
9
10 print('\nbirthday_month -->')
11 print(' value:', str(birthday_month))
12 print(' type:', type(birthday_month))
13 print(' id:', id(birthday_month))
14
15 print('\nbirthday_day -->')
16 print(' value:', str(birthday_day))
17 print(' type:', type(birthday_day))
18 print(' id:', id(birthday_day))
19
```


PARTICIPATION ACTIVITY 2.3.3: Objects basics.

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

Check Show answer

2) Write an expression that gives the identity of the variable num.

Check Show answer

2.4 Numeric types: Floating-point

Floating-point numbers and scientific notation

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

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

Figure 2.4.1: A program using float-type variables.

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

Note that print handles floating-point numbers straightforwardly.

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

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

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

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

PARTICIPATION ACTIVITY 2.4.1: Scientific notation.

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

Check Show answer

2) Type 7.2e-4 as a floating-point literal but not using scientific notation, with a

single digit before and five digits after the decimal point.

Check Show answer

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

Check Show answer

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

Check Show answer

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

Check Show answer

zyDE 2.4.1: Energy to mass conversion.

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

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

Load default template...

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

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

#Compute E = mc2.
energy_joules = mass_kg * (c_meters_per_sec**2)  # E = mc^2
print('Total energy released:', energy_joules, 'Joules')

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

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

```

0.1

Run

Overflow

Float-type objects have a limited range of values that can be represented. For a standard 32-bit installation of Python, the maximum floating-point value is approximately 1.8×10^{308}, and the minimum floating-point value is 2.3×10^{-308}. Assigning a floating-point value outside of this range generates an ***OverflowError***. ***Overflow*** occurs when a value is too large to be stored in the memory allocated by the interpreter. For example, the program in the figure below tries to store the value 2.0^{1024}, which causes an overflow error.

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

Figure 2.4.2: Float can overflow.

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

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

PARTICIPATION ACTIVITY 2.4.2: Floating-point versus integer.

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

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

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

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

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

5) The average number of kids per household.
   - ○ float

○ int

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

Finish the program to compute how many gallons of paint are needed to cover the given square feet of walls. Assume 1 gallon can cover 350.0 square feet. So gallons = the square feet divided by 350.0. If the input is 250.0, the output should be:

```
0.714285714286
```



```
gallons_paint = 0.0

wall_area = float(input())

# Assign gallons_paint below

''' Your solution goes here '''

print(gallons_paint)
```

Run

View solution *(Instructors only)*

Download student submissions

2.5 Arithmetic expressions

Basics

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

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

Table 2.5.1: Arithmetic operators.

Arithmetic operator	Description
+	The ***addition*** operator is ***+***, as in x + y.
-	The ***subtraction*** operator is ***-***, as in x - y. Also, the - operator is for ***negation***, as in -x + y, or x + -y.

*	The ***multiplication*** operator is *******, as in x * y.
/	The ***division*** operator is **/**, as in x / y.
**	The ***exponent*** operator is ********, as in x ** y (x to the power of y).

PARTICIPATION ACTIVITY 2.5.1: Expressions.

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

1) x + 1
   - Valid
   - Not valid

2) 2 * (x - y)
   - Valid
   - Not valid

3) x
   - Valid
   - Not valid

4) 2
   - Valid
   - Not valid

5) 2x
   - Valid
   - Not valid

6) 2 + (xy)
   - Valid
   - Not valid

7) y = x + 1
   - Valid
   - Not valid

PARTICIPATION ACTIVITY 2.5.2: Capturing behavior with an expression.

Does the expression correctly capture the intended behavior?

1) 6 plus num_items:

```
6 + num_items
```

   - Yes
   - No

2) 6 times num_items:

```
6 x num_items
```

   - Yes

No

3) total_days divided by 12:

```
total_days / 12
```

- Yes
- No

4) 5 times t:

```
5t
```

- Yes
- No

5) The negative of user_val:

```
-user_val
```

- Yes
- No

6) n factorial

```
n!
```

- Yes
- No

Evaluation of expressions

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

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

Table 2.5.2: Precedence rules for arithmetic operators.

Operator/Convention	Description	Explanation
( )	Items within parentheses are evaluated first.	In 2 * (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2.
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	In y = x * 2 / 3, the x * 2 is first evaluated, with the result then divided by 3.

	evaluated, evaluation occurs left to right.	

PARTICIPATION ACTIVITY 2.5.3: 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 2.5.4: 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_seconds / 4.184
```

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

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

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

[ ] **×** **—** **Spaces in variable names**

Replaced by ( )

Underscores

-

*

Reset

2.6 Python expressions

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

Figure 2.6.1: Expression example: Leasing cost.

```
Enter down payment: 500
Enter monthly payment: 300
Enter number of months: 60
Total cost: 18500
```

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

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

total_cost = down_payment + (payment_per_month * num_months)

print ('Total cost:', total_cost)
```


PARTICIPATION ACTIVITY 2.6.1: Simple program with an arithmetic expression.

Consider the example above.

1) Would removing the parentheses as below have yielded the same result?

```
down_payment + payment_per_month *
num_months
```

- ○ Yes
- ○ No

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

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

- ○ Yes
- ○ No

Style: Single space around operators

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

PARTICIPATION ACTIVITY 2.6.2: Single space around operators.

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

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

1) `houses_city = houses_block *10`

Check Show answer

2) `total = num1+num2+2`

Check Show answer

3) `num_balls=num_balls+1`

Check Show answer

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

Check Show answer

Compound operators

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

PARTICIPATION ACTIVITY 2.6.3: Compound operators.

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

Check Show answer

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

Check Show answer

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

Check Show answer

4) Rewrite the statement using a compound operator, or type: Not possible
`num_items = box_count + 1`

Check Show answer

No commas allowed

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

PARTICIPATION ACTIVITY 2.6.4: Assigning an integer literal.

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

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

- ○ True
- ○ False

CHALLENGE ACTIVITY 2.6.1: Computing an average.

Write a *single* statement that assigns avg_sales with the average of num_sales1, num_sales2, and num_sales3.

Sample output with inputs: 3 4 8

```
Average sales: 5.00
```



```
avg_sales = 0

num_sales1 = int(input())
num_sales2 = int(input())
num_sales3 = int(input())

''' Your solution goes here '''

print('Average sales: {:.2f}'.format(avg_sales))
```

Run

View solution *(Instructors only)*

Download student submissions

CHALLENGE ACTIVITY 2.6.2: Sphere volume.

Given sphere_radius and pi, compute the volume of a sphere and assign to sphere_volume.
Volume of sphere = (4.0 / 3.0) π r^3

Sample output with input: 1.0

```
Sphere volume: 4.19
```



```
pi = 3.14159
sphere_volume = 0.0

sphere_radius = float(input())

''' Your solution goes here '''

print('Sphere volume: {:.2f}'.format(sphere_volume))
```

Run

View solution *(Instructors only)*

Download student submissions

CHALLENGE ACTIVITY 2.6.3: Acceleration of gravity.

Compute the approximate acceleration of gravity for an object above the earth's surface, assigning accel_gravity with the result. The expression for the acceleration of gravity is: $(G * M) / (d^2)$, where G is the gravitational constant 6.673 x 10^{-11}, M is the mass of the earth 5.98 x 10^{24} (in kg), and d is the distance in meters from the Earth's center (stored in variable dist_center).

Sample output with input: 6.3782e6 (100 m above the Earth's surface at the equator)

```
Acceleration of gravity: 9.81
```

```
G = 6.673e-11
M = 5.98e24
accel_gravity = 0.0

dist_center = float(input())

''' Your solution goes here '''

print('Acceleration of gravity: {:.2f}'.format(accel_gravity))
```

Run

View solution *(Instructors only)*

Download student submissions

2.7 Division and modulo

Division: Integer rounding

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

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

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

- 20 // 10 is 2.
- 50 // 50 is 1.
- 5 // 10 is 0. (5/10 is 0 and the remainder 5 is thrown away).
- 5.0 // 2 is 2.0

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

PARTICIPATION ACTIVITY 2.7.1: Division and floored division.

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

1) `12 / 4`

Check Show answer

2) `5 / 10`

Check Show answer

3) `5.0 // 2`

Check Show answer

4) `100 / 0`

Check Show answer

Modulo (%)

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

Examples:

- 24 % 10 is 4. Reason: 24 / 10 is 2 with remainder 4.
- 50 % 50 is 0. Reason: 50 / 50 is 1 with remainder 0.
- 1 % 2 is 1. Reason: 1 / 2 is 0 with remainder 1.

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

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

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

Load default template...

```
75
```

Run

PARTICIPATION ACTIVITY 2.7.2: Modulo.

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

```
minutes = int(input('Enter minutes:\n'))
hours = minutes // 60
minutes_remaining = minutes % 60

print(minutes, 'minutes is', end=' ')
print(hours, 'hours and', end=' ')
print(minutes_remaining, 'minutes.\n', end=' '
```

1) `50 % 2`

Check Show answer

2) `51 % 2`

Check Show answer

3) `78 % 10`

Check Show answer

4) `596 % 10`

Check Show answer

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

Check Show answer

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

Start

Type the program's output.

```
x = 3
y = 2 * (x + 7)

print (x, y)
```

3 20

1 2 3 4

Check Next

View solution *(Instructors only)*

Modulo examples

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

Example 2.7.1: Getting digits.

Given a number, % and / can be used to get each digit. For a 3-digit number user_val like 927:

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

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

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

Example 2.7.2: Get prefix.

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

```
tmp_val = phone_num // 10000  # // 10000 shifts right by 4, so 936555.
prefix_num = tmp_val % 1000 # % 1000 gets the right 3 digits, so 555.
```

Dividing by a power of 10 shifts a value right. 321 // 10 is 32. 321 // 100 is 3.

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

PARTICIPATION ACTIVITY 2.7.3: Modulo examples.

1) Given a non-negative number x, which yields a number in the range 5 - 10?
   - ○ x % 5
   - ○ x % 10
   - ○ x % 11
   - ○ (x % 6) + 5

2) Given a non-negative number x, which expression has the range -10 to 10?
   - ○ x % -10
   - ○ (x % 21) - 10
   - ○ (x % 20) - 10

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

x % 10

- ○ x % 100
- ○ (x / / 10) % 10

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

- ○ x / 10000
- ○ x % 10000

CHALLENGE ACTIVITY 2.7.2: Compute change.

A cashier distributes change using the maximum number of five-dollar bills, followed by one-dollar bills. Write a single statement that assigns num_ones with the number of distributed one-dollar bills given amount_to_change. Hint: Use %.

Sample output with input: 19

```
Change for $ 19
3 five dollar bill(s) and 4 one dollar bill(s)
```

```
1 amount_to_change = int(input())
2
3 num_fives = amount_to_change // 5
4
5 ''' Your solution goes here '''
6
7 print('Change for $', amount_to_change)
8 print(num_fives, 'five dollar bill(s) and', num_ones, 'one dollar bill(s)')
```

Run

View solution *(Instructors only)*

Download student submissions

2.8 Module basics

Modules

The interactive Python interpreter allows a programmer to execute one line of code at a time. This method of programming is mostly used for very short programs or for practicing the language syntax. Instead, programmers

typically write Python program code in a file called a ***script***, and execute the code by passing the script as input to the Python interpreter.

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

Animation captions:

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

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

PARTICIPATION ACTIVITY 2.8.2: Importing modules.

Animation captions:

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

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

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

PARTICIPATION ACTIVITY 2.8.3: Basic modules.

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

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

Reset

Importing modules and executing scripts

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

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

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

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

```
# The baby_names.py module

print ('Initializing baby variables...')
baby_name1 = 'Ryder'
baby_name2 = 'Jess'
baby_weight1 = 5.1
baby_weight2 = 8.5

# Executes only if file run as a script (e.g., python
baby_names.py)
if __name__ == '__main__':
    print('Baby 1:', baby_name1, 'was born', baby_weight1,
'lbs')
    print('Baby 2:', baby_name2, 'was born', baby_weight2,
'lbs')

# A script favorite_child.py that imports and uses the
baby_names module.

import baby_names  # Importing the module executes the module
contents

print('My favorite child is', baby_names.baby_name1, '-')
print('I remember when he weighed only',
baby_names.baby_weight1, 'pounds.')
print('I love', baby_names.baby_name2, 'too, of course.')
```

```
$ python baby_names.py
Initializing baby
variables...
Baby 1: Ryder was born
5.1 lbs
Baby 2: Jess was born 8.5
lbs
```

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

PARTICIPATION ACTIVITY 2.8.4: Importing modules and executing scripts.

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

constants.py

```
# Gravitational constants for
various planets

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

if __name__ == '__main__':
    print('Earth constant:',
earth_g)
    print('Mars constant:',
mars_g)
```

fall_time.py

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

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

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


1) $ python constants.py

- ○ NO OUTPUT
- ○

Earth constant: 9.81
Mars constant: 3.71

- Height in meters:
  Earth: 1.4278431229270645 seconds
  Mars: 2.32181730106286 seconds

2) $ python fall_time.py

- NO OUTPUT
- Earth constant: 9.81
  Mars constant: 3.71
- Height in meters:
  Earth: 1.4278431229270645 seconds
  Mars: 2.32181730106286 seconds

2.9 Math module

The math module

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

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

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

```
import math

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

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

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

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

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

```
import math

base = float(input('Enter initial savings: '
print()

rate = float(input('Enter annual interest %
print()

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

total = base * math.pow(1 + (rate / 100), ye

print ('Savings after', years, 'years is', t
```

```
5000
3.5
20
```

Run

Commonly used functions

Commonly used functions from the math module are listed below. http://docs.python.org/3.7/library/math.html has a complete listing.

Table 2.9.1: Functions in the standard math module.

Function	Description		Function	Description
ceil	Round up value		fabs	Absolute value
factorial	factorial (3! = 3 * 2 * 1)		floor	Round down value
fmod	Remainder of division		fsum	Floating-point sum
exp	Exponential function e^x		log	Natural logarithm
pow	Raise to power		sqrt	Square root
acos	Arc cosine		asin	Arc sine
atan	Arc tangent		atan2	Arc tangent with two parameters
cos	Cosine		sin	Sine
hypot	Length of vector from origin		degrees	Convert from radians to degrees
radians	Convert degrees to radians		tan	Tangent
cosh	Hyperbolic cosine		sinh	Hyperbolic sine
gamma	Gamma function		erf	Error function
pi (constant)	Mathematical constant 3.141592...		e (constant)	Mathematical constant 2.718281...

PARTICIPATION ACTIVITY 2.9.1: Variable assignments with math functions.

Determine the final value of z.

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

Check Show answer

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

Check Show answer

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

Check Show answer

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

Check Show answer

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

Check Show answer

CHALLENGE ACTIVITY 2.9.1: Coordinate geometry.

Assign point_dist with the distance between point (x1, y1) and point (x2, y2). The calculation is: Distance = SquareRootOf($(x2 - x1)^2 + (y2 - y1)^2$).

Sample output with inputs: 1.0 2.0 1.0 5.0

```
Points distance: 3.0
```

```
import math

point_dist = 0.0

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

''' Your solution goes here '''

print('Points distance:', point_dist)
```

Run

View solution *(Instructors only)*

Download student submissions

CHALLENGE ACTIVITY 2.9.2: Tree height.

Simple geometry can compute the height of an object from the object's shadow length and shadow angle using the formula: tan(angleElevation) = treeHeight / shadowLength. Given the shadow length and angle of elevation, compute the tree height.

Sample output with inputs: 0.4 17.5

```
Tree height: 7.398881327917831
```

```
import math

tree_height = 0.0

angle_elev  = float(input())
shadow_len  = float(input())

''' Your solution goes here '''

print('Tree height:', tree_height)
```

Run

View solution *(Instructors only)*

Download student submissions

2.10 Representing text

Unicode

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

Table 2.10.1: Encoded text values.

Decimal	Character	Decimal	Character	Decimal	Character
32	space	64	@	96	`

33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

©zyBooks 03/05/20 10:21 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY 2.10.1: Unicode.

1)

What is the decimal encoding of the '{' character?

Check Show answer

Escape sequences

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

Table 2.10.2: Common escape sequences.

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

PARTICIPATION ACTIVITY 2.10.2: Escape sequences.

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

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

3) What is the output of `print('10...\n9...')`
   - 10...9...
   - 10...
     9...

10...\n9...

Raw strings and converting between an encoding and text

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

Figure 2.10.1: Ignoring escape characters with a raw string.


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

print(my_string)
print(my_raw_string)
```

```
This is a
 'normal' string

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

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

PARTICIPATION ACTIVITY 2.10.3: Using ord() to convert a character to the encoded value.

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

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

PARTICIPATION ACTIVITY 2.10.4: Using chr() to convert an encoded value to a character.

Type any number between 0 and 255 and observe the encoded value's character equivalent. Note that not all numbers will result a visible character.

Type a number (0-255): chr(66) Unicode char: **B**

PARTICIPATION ACTIVITY 2.10.5: Text.

1) Complete the code to output

\\

```
print(                    )
```

Check Show answer

2) Use a raw string literal to assign "C:\file.doc" to my_str.

```
my_str =
```

Check Show answer

Exploring further:

- Unicode HOWTO from the official Python documentation.

2.11 Additional practice: Number games

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 2.11.1: Number game.

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

Load default template...

```
num = int(input('Enter 2 digit number:\n'))

result = num * (3 * 7 * 13 * 37)

print(result)

```

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 2.11.2: Number game.

Load default templat

```
5
8
```

Run

```
# 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)

#Build num_out from its digits:
#num_out = ? + ? + ?   FINISH

# Note this line will generate an error until the above program is complete
```

2.12 LAB: Divide by x

Write a program using integers user_num and x as input, and output user_num divided by x three times.

Ex: If the input is:

```
2000
2
```

Then the output is:

```
1000 500 250
```

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

LAB ACTIVITY 2.12.1: LAB: Divide by x — 0 / 10

main.py — Load default template...

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

Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

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

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

2.13 LAB: Driving costs

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

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:
`print('{:.2f} {:.2f} {:.2f}'.format(your_value1, your_value2, your_value3))`

Ex: If the input is:

```
20.0
3.1599
```

Then the output is:

```
3.16 11.85 79.00
```

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

LAB ACTIVITY 2.13.1: LAB: Driving costs 0 / 10

main.py Load default template...

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


Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

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

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

2.14 LAB: Expression for calories burned during workout

The following equations estimate the calories burned when exercising (source):

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

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

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

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:
`print('Men: {:.2f} calories'.format(calories_man))`

Ex: If the input is:

```
49
155
148
60
```

Then the output is:

```
Women: 580.94 calories
Men: 891.47 calories
```

LAB ACTIVITY 2.14.1: LAB: Expression for calories burned during workout 0 / 10

main.py Load default template...

```
1 ''' Women: Calories = ((Age x 0.074) - (Weight x 0.05741) + (Heart Rate x 0.4472) - 20.4022) x Time /
2 ''' Men: Calories = ((Age x 0.2017) + (Weight x 0.09036) + (Heart Rate x 0.6309) - 55.0969) x Time / 4
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) → Output (shown below)

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

2.15 LAB: Using math functions

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

Output each floating-point value with two digits after the decimal point, which can be achieved as follows: `print('{:.2f} {:.2f} {:.2f} {:.2f}'.format(your_value1, your_value2, your_value3, your_value4))`

Ex: If the input is:

```
5.0
1.5
3.2
```


Then the output is:

```
172.47 361.66 3.50 13.13
```

LAB ACTIVITY 2.15.1: LAB: Using math functions 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 (shown below)

Program output displayed here

Lab statistics and submissions Show

Solution Show

Show

Tests

2.16 LAB: Musical note frequencies

On a piano, a key has a frequency, say f0. Each higher key (black or white) has a frequency of f0 * r^n, where n is the distance (number of keys) from that key, and r is $2^{(1/12)}$. Given an initial key frequency, output that frequency and the next 4 higher key frequencies.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows: `print('{:.2f} {:.2f} {:.2f} {:.2f} {:.2f}'.format(your_value1, your_value2, your_value3, your_value4, your_value5))`

Ex: If the input is:

```
440
```

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

```
440.00 466.16 493.88 523.25 554.37
```

Note: Use one statement to compute r = $2^{(1/12)}$ using the pow function (remember to import the math module). Then use that r in subsequent statements that use the formula fn = f0 * r^n with n being 1, 2, 3, and finally 4.

LAB ACTIVITY 2.16.1: LAB: Musical note frequencies 0 / 10

main.py Load default template...

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

Develop mode | Submit mode

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

Enter program input (optional)

If your code requires input values, provide them here.

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

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

2.17 LAB: Warm up: Variables, input, and type conversion

(1) Prompt the user to input an integer between 32 and 126, a float, a character, and a string, storing each into separate variables. Then, output those four values on a single line separated by a space. (Submit for 2 points).

Note: This zyLab outputs a newline after each user-input prompt. For convenience in the examples below, the user's input value is shown on the next line, but such values don't actually appear as output when the program runs.

```
Enter integer (32 - 126):
99
Enter float:
3.77
Enter character:
z
Enter string:
Howdy
99 3.77 z Howdy
```

(2) Extend to also output in reverse. (Submit for 1 point, so 3 points total).

```
Enter integer (32 - 126):
99
Enter float:
3.77
Enter character:
z
Enter string:
Howdy
99 3.77 z Howdy
Howdy z 3.77 99
```


(3) Extend to convert the integer to a character by using the 'chr()' function, and output that character. (Submit for 2 points, so 5 points total).

```
Enter integer (32 - 126):
99
Enter float:
3.77
Enter character:
z
```

```
Enter string:
Howdy
99 3.77 z Howdy
Howdy z 3.77 99
99 converted to a character is c
```

LAB ACTIVITY 2.17.1: LAB: Warm up: Variables, input, and type conversion 0 / 5

main.py

Load default template...

```
1 user_int = int(input('Enter integer (32 - 126):\n'))
2
3 # FIXME (1): Finish reading other items into variables, then output the four values on a single line s
4
5 # FIXME (2): Output the four values in reverse
6
7 # FIXME (3): Convert the integer to a characer, and output that character
```

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 (shown below)

Program output displayed here

Lab statistics and submissions Show

Solution Show

Tests Show

2.18 LAB*: Program: Cooking measurement converter

Output each floating-point value with two digits after the decimal point, which can be achieved as follows: `print('{:.2f}'.format(your_value))`

(1) Prompt the user for the number of cups of lemon juice, water, and agave nectar needed to make lemonade. Prompt the user to specify the number of servings the recipe yields. Output the ingredients and serving size. (Submit for 2 points).

Note: This zyLab outputs a newline after each user-input prompt. For convenience in the examples below, the user's input value is shown on the next line, but such values don't actually appear as output when the program runs.


```
Enter amount of lemon juice (in cups):
2
Enter amount of water (in cups):
16
Enter amount of agave nectar (in cups):
2.5
How many servings does this make?
6

Lemonade ingredients - yields 6.00 servings
2.00 cup(s) lemon juice
16.00 cup(s) water
2.50 cup(s) agave nectar
```

(2) Prompt the user to specify the desired number of servings. Adjust the amounts of each ingredient accordingly, and then output the ingredients and serving size. (Submit for 4 points, so 6 points total).

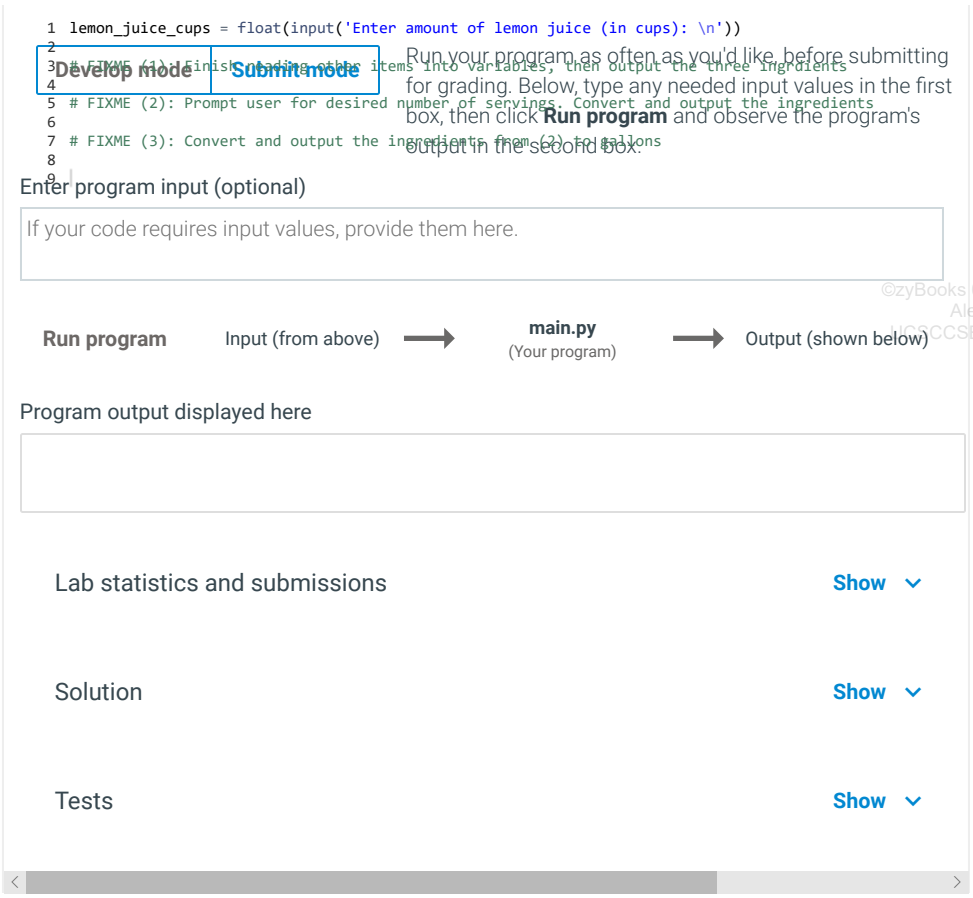
```
How many servings would you like to make?
48

Lemonade ingredients - yields 48.00 servings
16.00 cup(s) lemon juice
128.00 cup(s) water
20.00 cup(s) agave nectar
```

(3) Convert the ingredient measurements from (2) to gallons. Output the ingredients and serving size. Note: There are 16 cups in a gallon. (Submit for 2 points, so 8 points total).

```
Lemonade ingredients - yields 48.00 servings
1.00 gallon(s) lemon juice
8.00 gallon(s) water
1.25 gallon(s) agave nectar
```

LAB ACTIVITY 2.18.1: LAB*: Program: Cooking measurement converter 0 / 8

main.py Load default template...



2.19 LAB*: Program: Food receipt

Note: When accuracy is essential, floats are not used to represent currency due to rounding and accumulation errors. Python provides several primitives specifically developed to implement financial applications. However, these topics are beyond the scope of this lab.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:
`print('{:.2f}'.format(your_value))`

(1) Prompt the user to input a food item name, price, and quantity. Output an itemized receipt. (Submit for 2 points)

Note: This zyLab outputs a newline after each user-input prompt. For convenience in the examples below, the user's input value is shown on the next line, but such values don't actually appear as output when the program runs.

```
Enter food item name:
hot dog
Enter item price:
2.00
Enter item quantity:
5

RECEIPT
5 hot dog @ $2.00 = $10.00
Total cost: $10.00
```

(2) Extend the program to prompt the user for a second item. Output an itemized receipt. (Submit for 2 points, so 4 points total)

```
Enter food item name:
hot dog
Enter item price:
2.00
Enter item quantity:
5

RECEIPT
5 hot dog @ $2.00 = $10.00
Total cost: $10.00


Enter second food item name:
ice cream
Enter item price:
2.50
Enter item quantity:
4

RECEIPT
5 hot dog @ $2.00 = $10.00
4 ice cream @ $2.50 = $10.00
Total cost: $20.00
```


(3) Extend again to output a third receipt that adds a mandatory 15% gratuity to the total cost. Output the total cost, the cost of gratuity, and the grand total. (Submit for 3 points, so 7 points total)

```
Enter food item name:
hot dog
Enter item price:
2.00
Enter item quantity:
5

RECEIPT
5 hot dog @ $2.00 = $10.00
Total cost: $10.00


Enter second food item name:
ice cream
Enter item price:
2.50
Enter item quantity:
4

RECEIPT
5 hot dog @ $2.00 = $10.00
4 ice cream @ $2.50 = $10.00
Total cost: $20.00

15% gratuity: $3.00
Total with tip: $23.00
```


LAB ACTIVITY 2.19.1: LAB*: Program: Food receipt 0 / 7

main.py

```
item_name = input('Enter food item name:\n')

# FIXME (1): Finish reading item price and quantity, then output a receipt

# FIXME (2): Read in a second food item name, price, and quantity, then output a second receipt

# FIXME (3): Add a gratuity and total with tip to the second receipt
```


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 (shown below)

Program output displayed here

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

©zyBooks 03/05/20 10:21 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020