20.1 Variables and assignments

Remembering a value

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

PARTICIPATION ACTIVITY 20.1.1: People on bus.

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

Start

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

num_people: 5

1 2 3 4 5

Check Next

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 20.1.2: 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 | 20.1.3: Valid assignment statements.

Indicate which assignment statements are valid.

1) x = 1

- Valid
- Invalid

2) x = y

- Valid
- Invalid

3) x = y + 2

- Valid
- Invalid

4) x + 1 = 3

- Valid
- Invalid

5) x + y = y + x

- Valid
- Invalid

PARTICIPATION ACTIVITY 20.1.4: Variables and assignment statements.

Given variables x, y, and z.

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

Check Show answer

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

Check Show answer

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

Check Show answer

PARTICIPATION ACTIVITY | 20.1.5: Trace the variable value.

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

Start


```
int x = 6;
int y = 0;
int z = 3;
x = 2;
y = 7;
z = 9;
x = 0;
```

x is

6 2 0

y is

6 0 7

z is

9 1 3

1 2 3 4

Check Next

CHALLENGE ACTIVITY | 20.1.1: Enter the output of the variable assignments.

422102.2723990.qx3zqy7

Start

Type the program's output

```
x = 7
y = 5

print(x, y)
```

7 5

1 2 3 4 5

Check Next

Assignments with variable on left and right

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

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

PARTICIPATION ACTIVITY | 20.1.6: A variable may appear on the left and right of an assignment statement.

Animation content:

```
undefined
```

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 | 20.1.7: Variable on both sides.

Indicate the value of x after the statements execute.

1) x = 5
x = x + 7

Check Show answer

2) x = 2
y = 3

x = x * y
x = x * y

Check Show answer

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

Check Show answer

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

Check Show answer

CHALLENGE ACTIVITY 20.1.2: 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
```

422102.2723990.qx3zqy7

```
total_coins = 0

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

''' Your solution goes here '''

print(total_coins)
```

Run

CHALLENGE ACTIVITY | 20.1.3: 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
```

422102.2723990.qx3zqy7

```
cell_count = int(input())

''' Your solution goes here '''

print(cell_count)
```

Run

20.2 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 or underscore), hi there (has a space), and cat$ (has a symbol other than a letter, digit, or underscore).

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

Certain words like "and" or "True" cannot be used as names. ***Reserved words***, or ***keywords***, are words that are part of the language, and 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 20.2.1: Valid names.

Which of the following are valid names?

1) numCars
   - Valid
   - Invalid

2) num_cars1
   - Valid
   - Invalid

3) _num_cars
   - Valid
   - Invalid

4) ___numcars2
   - Valid
   - Invalid

5) num cars
   - Valid
   - Invalid

6) 3rd_place

- Valid
- Invalid

7) third_place_
- Valid
- Invalid

8) third_place!
- Valid
- Invalid

9) output
- Valid
- Invalid

10) return_copy
- Valid
- Invalid

Style guidelines for identifiers

A good practice when naming variables is to use all lowercase letters and to place underscores between words. This lowercase and underscore convention for naming variables originates from 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 20.2.1: 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 20.2.2: Python 3 reserved keywords.

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


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

PARTICIPATION ACTIVITY | 20.2.2: Python 3 name validator.

Use the tool below to test valid and invalid names.

Enter an identifier: ____________ **Validate**

20.3 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 20.3.1: 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 20.3.2: 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 20.3.1: Printing displays an object's value.

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

```
4
5
```

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

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

Figure 20.3.2: 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 20.3.3: 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 20.3.1: Experimenting with objects.

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

Load default template...

Run

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

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

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

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


PARTICIPATION ACTIVITY 20.3.3: Objects basics.

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

Check Show answer

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

Check Show answer

20.4 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 20.4.1: 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 | 20.4.1: Scientific notation.

1) Type 1.0e-4 as a floating-point literal with a single digit before and four digits after the decimal point. Note: Do not use scientific notation.

Check Show answer

2) Type 7.2e-4 as a floating-point literal with a single digit before and five digits after the decimal point. Note: Do not use scientific notation.

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 20.4.1: Energy to mass conversion.

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

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

Load default ter

```
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 = mc^2.
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')
```

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 1.8x10^{308}, and the minimum floating-point value is 2.3x10^{-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 20.4.2: 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 20.4.2: Floating-point versus integer.

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

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

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

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

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

int

5) The average number of kids per household.

float

int

Manipulating floating-point output

Some floating-point numbers have many digits after the decimal point. Ex: Irrational numbers (Ex: 3.14159265359...) and repeating decimals (Ex: 4.33333333...) have an infinite number of digits after the decimal. By default, most programming languages output at least 5 digits after the decimal point. But for many simple programs, this level of detail is not necessary. A common approach is to output floating-point numbers with a specific number of digits after the decimal to reduce complexity or produce a certain numerical type (Ex: Representing currency with two digits after the decimal). The syntax for outputting the float myFloat with two digits after the decimal point is `print(f'{myFloat:.2f}')`

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

PARTICIPATION ACTIVITY | 20.4.3: Reducing the output of Pi.

Animation content:

`undefined`

Animation captions:

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

PARTICIPATION ACTIVITY | 20.4.4: Reducing floating-point output.

1) Which of the following arguments completes print() to output two digits after the decimal point?
`print(f'{(7.0 / 3.0)_____)`

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

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

- 0
- 0.1
- 0.13

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

- 9.136
- 9.135
- 9.14

CHALLENGE ACTIVITY 20.4.1: 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:

```
0.7142857142857143
```

Note: Do not format the output.

422102.2723990.qx3zqy7


```
wall_area = float(input())

# Assign gallons_paint below

''' Your solution goes here '''

print(gallons_paint)
```

Run

20.5 Python expressions

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

Figure 20.5.1: Expression example: Leasing cost.

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

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

total_cost = down_payment + (payment_per_month *
num_months)

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

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

PARTICIPATION ACTIVITY 20.5.1: 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?

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

- Yes
- No

Style: Single space around operators

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

PARTICIPATION ACTIVITY 20.5.2: Single space around operators.

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

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

1) `houses_city = houses_block *10`

Check Show answer

2) `total = num1+num2+2`

Check Show answer

3) `num_balls=num_balls+1`

Check Show answer

4) `num_entries =`

(user_val+1)*2

Check Show answer

Compound operators

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

Table 20.5.1: Compound operators.

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

PARTICIPATION ACTIVITY | 20.5.3: 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 20.5.4: Assigning an integer literal.

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

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


- True
- False

CHALLENGE ACTIVITY 20.5.1: 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
```

422102.2723990.qx3zqy7

```
avg_sales = 0

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

''' Your solution goes here '''

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

Run

CHALLENGE ACTIVITY 20.5.2: Sphere volume.

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

Sample output with input: 1.0

```
Sphere volume: 4.19
```

422102.2723990.qx3zqy7

Run

CHALLENGE ACTIVITY | 20.5.3: 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
```

422102.2723990.qx3zqy7

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

dist_center = float(input())

''' Your solution goes here '''

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

Run

20.6 Division and modulo

Division: Integer rounding

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

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

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

- 20 // 10 is 2.
- 50 // 50 is 1.
- 5 // 10 is 0. (5/10 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 | 20.6.1: Division and floor division.

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

1) `12 / 4`

Check Show answer

2) `5 / 10`

Check Show answer

3) `5.0 // 2`

Check Show answer

4) `100 / 0`

Check Show answer

Modulo (%)

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

Examples:

- 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 20.6.1: Example using expressions: Minutes to hours/minutes.

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

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

Load default template...

```
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=
```

75

Run

PARTICIPATION ACTIVITY 20.6.2: Modulo.

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

1) `50 % 2`

Check Show answer

2) `51 % 2`

Check Show answer

3) `78 % 10`

Check Show answer

4) `596 % 10`

Check Show answer

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

Check Show answer

CHALLENGE ACTIVITY 20.6.1: Enter the output of the integer expressions.

422102.2723990.qx3zqy7

Start

Type the program's output

```
x = 10
y = x / 4

print(y)
```

2.5

1 2

Check Next

Modulo examples

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

Example 20.6.1: 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 20.6.2: 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. Ex: 321 // 10 is 32. Ex: 321 // 100 is 3.

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

PARTICIPATION ACTIVITY | 20.6.3: Modulo examples.

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

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

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

- x % -10

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

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

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

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

- x / 10000
- x % 10000

CHALLENGE ACTIVITY 20.6.2: 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)
```

422102.2723990.qx3zqy7

```
amount_to_change = int(input())

num_fives = amount_to_change // 5

''' Your solution goes here '''

print('Change for $', amount_to_change)
print(num_fives, 'five dollar bill(s) and', num_ones, 'one dollar bill(s)')
```

Run

20.7 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 20.7.1: 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 20.7.2: 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 | 20.7.3: Basic modules.

If unable to drag and drop, refresh the page.

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

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

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

Figure 20.7.1: Checking if a file was executed as a script.

```
# The pet_names.py module

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

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

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

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

import pet_names  # Importing the module executes
the module contents

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

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


PARTICIPATION ACTIVITY | 20.7.4: 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

20.8 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 20.8.1: 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 20.8.1: 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

Load default template...

```
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), y

print ('Savings after', years, 'years is',

```

```
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 20.8.1: Functions in the standard math module.

Function	Description		Function	Description
Number representation and theoretic functions				
ceil(x)	Round up value		fabs(x)	Absolute value
factorial(x)	factorial (3! = 3 * 2 * 1)		floor(x)	Round down value
fmod(x, y)	Remainder of division		fsum(x)	Floating-point sum of a range, list, or array.
Power, exponential, and logarithmic functions				
exp(x)	Exponential function e^x		log(x, (base))	Natural logarithm; base is optional
pow(x, y)	Raise x to power y		sqrt(x)	Square root
Trigonometric functions				
acos(x)	Arc cosine		asin(x)	Arc sine
atan(x)	Arc tangent		atan2(y, x)	Arc tangent with two parameters
cos(x)	Cosine		sin(x)	Sine
hypot(x1, x2, x3, ..., xn)	Length of vector from origin		degrees(x)	Convert from radians to degrees
radians(x)	Convert degrees to radians		tan(x)	Tangent
cosh(x)	Hyperbolic cosine		sinh(x)	Hyperbolic sine
Complex number functions				
gamma(x)	Gamma function		erf(x)	Error function
Mathematical constants				
pi (constant)	Mathematical constant 3.141592...		e (constant)	Mathematical constant 2.718281...

PARTICIPATION ACTIVITY | 20.8.1: 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 20.8.1: Math functions.

422102.2723990.qx3zqy7

Start

Type the program's output


```
import math

x = math.sqrt(16.0)
print(x)
```

4.0

1 2 3 4

Check Next

CHALLENGE ACTIVITY 20.8.2: Math functions.

422102.2723990.qx3zqy7

Start

Compute: $z = \sqrt{x * y}$

```
import math

x = float(input())
y = float(input())

''' Your code goes here '''

print(round(z, 2)) # This will output only 2 decimal places.

```


1 2 3

Check Next Show solution

CHALLENGE ACTIVITY | 20.8.3: Using math functions to calculate the distance between two points

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

Sample output with inputs: 1.0 2.0 1.0 5.0

```
Points distance: 3.0
```

422102.2723990.qx3zqy7

```
import math

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

''' Your solution goes here '''

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

Run

20.9 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 20.9.1: 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

©zyBooks 12/15/22 00:40 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

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	_		

PARTICIPATION ACTIVITY 20.9.1: 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 20.9.2: 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 20.9.2: 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 20.9.1: 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 20.9.3: 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 20.9.4: Using chr() to convert an encoded value to a character.

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

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

ASCII char: Z

CHALLENGE ACTIVITY 20.9.1: Enter the output of the print() statements.

422102.2723990.qx3zqy7

Start

Type the program's output

```
print('The name of the dog is "Ruby".')
```

The name of the dog

1 2 3 4

Check Next

PARTICIPATION ACTIVITY 20.9.5: Text.

1) Complete the code to output

```
\\
```

```
print(
)
```

Check Show answer

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

```
my_str =
```

Check Show answer

Exploring further:

- Unicode HOWTO from the official Python documentation.