codecademy

Hello World

Comments

A comment is a piece of text within a program that is not executed. It can be used to provide additional information to aid in understanding the code.

The `#` character is used to start a comment and it continues until the end of the line.

```
# Comment on a single line

user = "JDoe" # Comment after code
```

Arithmetic Operations

Python supports different types of arithmetic operations that can be performed on literal numbers, variables, or some combination. The primary arithmetic operators are:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `%` for modulus (returns the remainder)
- `**` for exponentiation

```
# Arithmetic operations

result = 10 + 30
result = 40 - 10
result = 50 * 5
result = 16 / 4
result = 25 % 2
result = 5 ** 3
```

Plus-Equals Operator +=

The plus-equals operator `+=` provides a convenient way to add a value to an existing variable and assign the new value back to the same variable. In the case where the variable and the value are strings, this operator performs string concatenation instead of addition. The operation is performed in-place, meaning that any other variable which points to the variable being updated will also be updated.

```
# Plus-Equal Operator

counter = 0
counter += 10

# This is equivalent to

counter = 0
counter = counter + 10

# The operator will also perform string
concatenation

message = "Part 1 of message "
message += "Part 2 of message"
```

Variables

A variable is used to store data that will be used by the program. This data can be a number, a string, a Boolean, a list or some other data type. Every variable has a name which can consist of letters, numbers, and the underscore character `_` .

The equal sign `=` is used to assign a value to a variable. After the initial assignment is made, the value of a variable can be updated to new values as needed.

```
# These are all valid variable names and assignment

user_name = "@sonnynomnom"
user_id = 100
verified = False

# A variable's value can be changed after assignment

points = 100
points = 120
```

Modulo Operator %

A modulo calculation returns the remainder of a division between the first and second number. For example:

- The result of the expression `4 % 2` would result in the value 0, because 4 is evenly divisible by 2 leaving no remainder.
- The result of the expression `7 % 3` would return 1, because 7 is not evenly divisible by 3, leaving a remainder of 1.

```
# Modulo operations

zero = 8 % 4

nonzero = 12 % 5
```

Integers

An integer is a number that can be written without a fractional part (no decimal). An integer can be a positive number, a negative number or the number 0 so long as there is no decimal portion.

The number `0` represents an integer value but the same number written as `0.0` would represent a floating point number.

```
# Example integer numbers

chairs = 4
tables = 1
broken_chairs = -2
sofas = 0

# Non-integer numbers

lights = 2.5
left_overs = 0.0
```

String Concatenation

Python supports the joining (concatenation) of strings together using the `+` operator. The `+` operator is also used for mathematical addition operations. If the parameters passed to the `+` operator are strings, then concatenation will be performed. If the parameter passed to `+` have different types, then Python will report an error condition. Multiple variables or literal strings can be joined together using the `+` operator.

```
# String concatenation

first = "Hello "
second = "World"

result = first + second

long_result = first + second + "!"
```

Errors

The Python interpreter will report errors present in your code. For most error cases, the interpreter will display the line of code where the error was detected and place a caret character `^` under the portion of the code where the error was detected.

```
if False ISNOTEQUAL True:
                 ^
SyntaxError: invalid syntax
```

ZeroDivisionError

A ZeroDivisionError is reported by the Python interpreter when it detects a division operation is being performed and the denominator (bottom number) is 0. In mathematics, dividing a number by zero has no defined value, so Python treats this as an error condition and will report a ZeroDivisionError and display the line of code where the division occurred. This can also happen if a variable is used as the denominator and its value has been set to or changed to 0.

```
numerator = 100
denominator = 0
bad_results = numerator / denominator

ZeroDivisionError: division by zero
```

Strings

A string is a sequence of characters (letters, numbers, whitespace or punctuation) enclosed by quotation marks. It can be enclosed using either the double quotation mark `"` or the single quotation mark `'`. If a string has to be broken into multiple lines, the backslash character `\` can be used to indicate that the string continues on the next line.

```
user = "User Full Name"
game = 'Monopoly'

longer = "This string is broken up \
over multiple lines"
```

SyntaxError

A SyntaxError is reported by the Python interpreter when some portion of the code is incorrect. This can include misspelled keywords, missing or too many brackets or parenthesis, incorrect operators, missing or too many quotation marks, or other conditions.

```
age = 7 + 5 = 4

File "<stdin>", line 1
SyntaxError: can't assign to operator
```

NameError

A NameError is reported by the Python interpreter when it detects a variable that is unknown. This can occur when a variable is used before it has been assigned a value or if a variable name is spelled differently than the point at which it was defined. The Python interpreter will display the line of code where the NameError was detected and indicate which name it found that was not defined.

```
misspelled_variable_name

NameError: name
'misspelled_variable_name' is not defined
```

Floating Point Numbers

Python variables can be assigned different types of data. One supported data type is the floating point number. A floating point number is a value that contains a decimal portion. It can be used to represent numbers that have fractional quantities. For example, `a = 3/5` can not be represented as an integer, so the variable `a` is assigned a floating point value of `0.6`.

```
# Floating point numbers

pi = 3.14159
meal_cost = 12.99
tip_percent = 0.20
```

`print()` Function

The `print()` function is used to output text, numbers, or other printable information to the console. It takes one or more arguments and will output each of the arguments to the console separated by a space. If no arguments are provided, the `print()` function will output a blank line.

```
print("Hello World!")

print(100)

pi = 3.14159
print(pi)
```

codecademy

Python: Control Flow

or Operator

The Python `or` operator combines two Boolean expressions and evaluates to `True` if at least one of the expressions returns `True`. Otherwise, if both expressions are `False`, then the entire expression evaluates to `False`.

```
True or True      # Evaluates to True
True or False     # Evaluates to True
False or False    # Evaluates to False
1 < 2 or 3 < 1    # Evaluates to True
3 < 1 or 1 > 6    # Evaluates to False
1 == 1 or 1 < 2   # Evaluates to True
```

`elif` Statement

The Python `elif` statement allows for continued checks to be performed after an initial `if` statement. An `elif` statement differs from the `else` statement because another expression is provided to be checked, just as with the initial `if` statement. If the expression is `True`, the indented code following the `elif` is executed. If the expression evaluates to `False`, the code can continue to an optional `else` statement. Multiple `elif` statements can be used following an initial `if` to perform a series of checks. Once an `elif` expression evaluates to `True`, no further `elif` statements are executed.

```
# elif Statement

pet_type = "fish"

if pet_type == "dog":
  print("You have a dog.")
elif pet_type == "cat":
  print("You have a cat.")
elif pet_type == "fish":
  # this is performed
  print("You have a fish")
else:
  print("Not sure!")
```

Handling Exceptions in Python

A `try` and `except` block can be used to handle error in code block. Code which may raise an error can be written in the `try` block. During execution, if that code block raises an error, the rest of the `try` block will cease executing and the `except` code block will execute.

```
def check_leap_year(year):
  is_leap_year = False
  if year % 4 == 0:
    is_leap_year = True

try:
  check_leap_year(2018)
  print(is_leap_year)
  # The variable is_leap_year is declared
inside the function
except:
  print('Your code raised an error!')
```

Equal Operator ==

The equal operator, `==`, is used to compare two values, variables or expressions to determine if they are the same.
If the values being compared are the same, the operator returns `True`, otherwise it returns `False`.
The operator takes the data type into account when making the comparison, so a string value of `"2"` is *not* considered the same as a numeric value of `2`.

```
# Equal operator

if 'Yes' == 'Yes':
  # evaluates to True
  print('They are equal')

if (2 > 1) == (5 < 10):
  # evaluates to True
  print('Both expressions give the same result')

c = '2'
d = 2

if c == d:
  print('They are equal')
else:
  print('They are not equal')
```

Not Equals Operator !=

The Python not equals operator, `!=`, is used to compare two values, variables or expressions to determine if they are NOT the same. If they are NOT the same, the operator returns `True`. If they are the same, then it returns `False`.
The operator takes the data type into account when making the comparison so a value of `10` would NOT be equal to the string value `"10"` and the operator would return `True`. If expressions are used, then they are evaluated to a value of `True` or `False` before the comparison is made by the operator.

```
# Not Equals Operator

if "Yes" != "No":
  # evaluates to True
  print("They are NOT equal")

val1 = 10
val2 = 20

if val1 != val2:
  print("They are NOT equal")

if (10 > 1) != (10 > 1000):
  # True != False
  print("They are NOT equal")
```

Comparison Operators

In Python, *relational operators* compare two values or expressions. The most common ones are:

- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal too

If the relation is sound, then the entire expression will evaluate to `True`. If not, the expression evaluates to `False`.

```
a = 2
b = 3
a < b  # evaluates to True
a > b  # evaluates to False
a >= b # evaluates to False
a <= b # evaluates to True
a <= a # evaluates to True
```

`if` Statement

The Python `if` statement is used to determine the execution of code based on the evaluation of a Boolean expression.

- If the `if` statement expression evaluates to `True`, then the indented code following the statement is executed.
- If the expression evaluates to `False` then the indented code following the `if` statement is skipped and the program executes the next line of code which is indented at the same level as the `if` statement.

```
# if Statement

test_value = 100

if test_value > 1:
  # Expression evaluates to True
  print("This code is executed!")

if test_value > 1000:
  # Expression evaluates to False
  print("This code is NOT executed!")

print("Program continues at this point.")
```

`else` Statement

The Python `else` statement provides alternate code to execute if the expression in an `if` statement evaluates to `False`.
The indented code for the `if` statement is executed if the expression evaluates to `True`. The indented code immediately following the `else` is executed only if the expression evaluates to `False`. To mark the end of the `else` block, the code must be unindented to the same level as the starting `if` line.

```
# else Statement

test_value = 50

if test_value < 1:
  print("Value is < 1")
else:
  print("Value is >= 1")

test_string = "VALID"

if test_string == "NOT_VALID":
  print("String equals NOT_VALID")
else:
  print("String equals something else!")
```

and Operator

The Python `and` operator performs a Boolean comparison between two Boolean values, variables, or expressions. If both sides of the operator evaluate to `True` then the `and` operator returns `True`. If either side (or both sides) evaluates to `False`, then the `and` operator returns `False`. A non-Boolean value (or variable that stores a value) will always evaluate to `True` when used with the `and` operator.

```
True and True      # Evaluates to True
True and False     # Evaluates to False
False and False    # Evaluates to False
1 == 1 and 1 < 2   # Evaluates to True
1 < 2 and 3 < 1    # Evaluates to False
"Yes" and 100      # Evaluates to True
```

Boolean Values

Booleans are a data type in Python, much like integers, floats, and strings. However, booleans only have two values:

- `True`
- `False`

Specifically, these two values are of the `bool` type. Since booleans are a data type, creating a variable that holds a boolean value is the same as with other data types.

```
is_true = True
is_false = False

print(type(is_true))
# will output: <class 'bool'>
```

not Operator

The Python Boolean `not` operator is used in a Boolean expression in order to evaluate the expression to its inverse value. If the original expression was `True`, including the `not` operator would make the expression `False`, and vice versa.

```
not True      # Evaluates to False
not False     # Evaluates to True
1 > 2         # Evaluates to False
not 1 > 2     # Evaluates to True
1 == 1        # Evaluates to True
not 1 == 1    # Evaluates to False
```

codecademy

Python: Functions

Returning Multiple Values

Python functions are able to return multiple values using one `return` statement. All values that should be returned are listed after the `return` keyword and are separated by commas.

In the example, the function `square_point()` returns `x_squared`, `y_squared`, and `z_squared`.

```
def square_point(x, y, z):
  x_squared = x * x
  y_squared = y * y
  z_squared = z * z
  # Return all three values:
  return x_squared, y_squared, z_squared

three_squared, four_squared, five_squared
= square_point(3, 4, 5)
```

The Scope of Variables

In Python, a variable defined inside a function is called a local variable. It cannot be used outside of the scope of the function, and attempting to do so without defining the variable outside of the function will cause an error.

In the example, the variable `a` is defined both inside and outside of the function. When the function `f1()` is implemented, `a` is printed as `2` because it is locally defined to be so. However, when printing `a` outside of the function, `a` is printed as `5` because it is implemented outside of the scope of the function.

```
a = 5

def f1():
  a = 2
  print(a)

print(a)   # Will print 5
f1()       # Will print 2
```

Function Parameters

Sometimes functions require input to provide data for their code. This input is defined using *parameters*. *Parameters* are variables that are defined in the function definition. They are assigned the values which were passed as arguments when the function was called, elsewhere in the code.

For example, the function definition defines parameters for a character, a setting, and a skill, which are used as inputs to write the first sentence of a book.

```
def write_a_book(character, setting,
special_skill):
  print(character + " is in " +
        setting + " practicing her " +
        special_skill)
```

Multiple Parameters

Python functions can have multiple *parameters*. Just as you wouldn't go to school without both a backpack and a pencil case, functions may also need more than one input to carry out their operations.

To define a function with multiple parameters, parameter names are placed one after another, separated by commas, within the parentheses of the function definition.

```
def ready_for_school(backpack,
pencil_case):
  if (backpack == 'full' and pencil_case
== 'full'):
    print ("I'm ready for school!")
```

Returning Value from Function

A `return` keyword is used to return a value from a Python function. The value returned from a function can be assigned to a variable which can then be used in the program.

In the example, the function `check_leap_year` returns a string which indicates if the passed parameter is a leap year or not.

```
def check_leap_year(year):
  if year % 4 == 0:
    return str(year) + " is a leap year."
  else:
    return str(year) + " is not a leap
year."

year_to_check = 2018
returned_value
= check_leap_year(year_to_check)
print(returned_value) # 2018 is not
a leap year.
```

Functions

Some tasks need to be performed multiple times within a program. Rather than rewrite the same code in multiple places, a function may be defined using the `def` keyword. Function definitions may include parameters, providing data input to the function.

Functions may return a value using the `return` keyword followed by the value to return.

```
# Define a function my_function() with
parameter x

def my_function(x):
  return x + 1

# Invoke the function

print(my_function(2))      # Output: 3
print(my_function(3 + 5))  # Output: 9
```

Function Indentation

Python uses indentation to identify blocks of code. Code within the same block should be indented at the same level. A Python function is one type of code block. All code under a function declaration should be indented to identify it as part of the function. There can be additional indentation within a function to handle other statements such as `for` and `if` so long as the lines are not indented less than the first line of the function code.

```
# Indentation is used to identify code
blocks

def testfunction(number):
  # This code is part of testfunction
  print("Inside the testfunction")
  sum = 0
  for x in range(number):
    # More indentation because 'for' has
a code block
    # but still part of he function
    sum += x
  return sum
print("This is not part of testfunction")
```

Calling Functions

Python uses simple syntax to use, invoke, or *call* a preexisting function. A function can be called by writing the name of it, followed by parentheses.
For example, the code provided would call the `doHomework()` method.

```
doHomework()
```

Global Variables

A variable that is defined outside of a function is called a global variable. It can be accessed inside the body of a function.

In the example, the variable `a` is a global variable because it is defined outside of the function `prints_a`. It is therefore accessible to `prints_a`, which will print the value of `a`.

```
a = "Hello"

def prints_a():
  print(a)

# will print "Hello"
prints_a()
```

Parameters as Local Variables

Function parameters behave identically to a function's local variables. They are initialized with the values passed into the function when it was called.
Like local variables, parameters cannot be referenced from outside the scope of the function.

In the example, the parameter `value` is defined as part of the definition of `my_function`, and therefore can only be accessed within `my_function`. Attempting to print the contents of `value` from outside the function causes an error.

```
def my_function(value):
  print(value)

# Pass the value 7 into the function
my_function(7)

# Causes an error as `value` no longer exists
print(value)
```

Function Arguments

Parameters in python are variables – placeholders for the actual values the function needs. When the function is *called*, these values are passed in as *arguments*.
For example, the arguments passed into the function `.sales()` are the "The Farmer's Market", "toothpaste", and "$1" which correspond to the parameters `grocery_store`, `item_on_sale`, and `cost`.

```
def sales(grocery_store, item_on_sale, cost):
  print(grocery_store + " is selling " + item_on_sale + " for " + cost)

sales("The Farmer's Market", "toothpaste", "$1")
```

Function Keyword Arguments

Python functions can be defined with named arguments which may have default values provided. When function arguments are passed using their names, they are referred to as keyword arguments. The use of keyword arguments when calling a function allows the arguments to be passed in any order — *not* just the order that they were defined in the function. If the function is invoked without a value for a specific argument, the default value will be used.

```
def findvolume(length=1, width=1,
depth=1):
  print("Length = " + str(length))
  print("Width = " + str(width))
  print("Depth = " + str(depth))
  return length * width * depth;

findvolume(1, 2, 3)
findvolume(length=5, depth=2, width=4)
findvolume(2, depth=3, width=4)
```