Learn Python 3 : Syntax

codecademy

Python `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)
```

Python Comments

A comment is a piece of text within a program that is *not* executed as part of the program. 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"  # End of line comment after code
```

Python 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"
userID = 100
prompt21 = "Enter your user name"
existing_user = False
another_userID = userID

# A variable's value can be changed after assignment

score = 100
score = 120
```

Python Integers

Python variables can be assigned different types of data. One supported data type is the integer. An integer is a number which 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
```

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 which contains a decimal portion. Floating point numbers are used to represent numbers which have fractional quantities. For example, `a = 3/5` can not be represented as an integer so the variable `a` is assigned the floating point value `0.6`.

```
# Floating point numbers
percentage = 0.75
pi = 3.14159
meal_cost = 8.49
```

Python Arithmetic Operations

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

- `+` for addition
- `-` for subtraction
- `/` for division
- `*` for multiplication

```
# Arithmetic operations

result = 100 + 300
result = 40 - 10
result = 50 * 5
result = 16 / 4

my_function(3 * 1.5)
```

Modulo Operator `%`

Python supports an operator to perform the modulo calculation. 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
```

Exponentiation

In addition to the basic operations of addition, subtraction, multiplication and division, Python supports an operator for exponentiation. That operator is written with two asterisks like so `**`. The format for exponentiation in Python is a number or variable followed by the operator `**` followed by a number or variable which represents the power to raise the number. Both the number and the power can be integer or floating point values.

```
# Exponential operator

result1 = 2 ** 4

power = 2
square = 2 ** power

half = 0.5
result2 = square ** half
```

Python Integer Division

Python 3 will automatically convert integer numbers to floating-point before performing division. This behavior is changed from Python 2 where integer numbers were NOT automatically converted. In Python 2, dividing by an integer number performed integer division, where the fractional part (remainder) of the result is discarded. To perform regular division in Python 2, use of the `float()` function or a literal floating point value was required to force division to produce a floating point result.

```
# Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21)
result = 5 / 3
print(result)
# 1.6666666666666667

# Python 2.7.10 (default, Oct 6 2017, 22:29)
result = 5 / 3
print result
# 1
result = 5 / float(3)
print result
# 1.66666666667
```

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"
```

Python 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"
```

Python 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. The concatenation process does not add any whitespace between the strings that are joined.

```
# String concatenation

first = "Hello "
second = "World"

result = first + second

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

Error Notification

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
```

Python 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
```

Python 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
```

Python 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
```

Learn Python 3 : Functions

codecademy

Python 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.

In the example, a function `my_function` is defined with the parameter `x`. The function returns a value that takes the parameter and adds 1 to it. It is then invoked multiple times with different input values.

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

def my_function(x):
  return x + 1

# invoke our function

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

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()
```

Defining Functions

A developer can create or *define* his or her own function in Python. To do so, the keyword `def` is followed by the name of the function, parentheses, and a colon. The body of the function, or the code for what the function will actually do, comes after the colon on indented lines.

```
def doHomework():
  # function body goes here
```

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 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)
```

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.
```

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")
```

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)
```

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 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")
```

Global Variables in Python

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()
```

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)
```

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)
```