Apply Python

LU2 – Develop Python Concept

NDAYISENGA Gilbert
UOK

Python syntax

Every language has a different syntax for displaying or printing something on the screen

We can print data on the terminal by simply using the print statement “**print(data)**”.

Let’s try printing “Hello World” on the terminal

```
print ('Hello World')
```

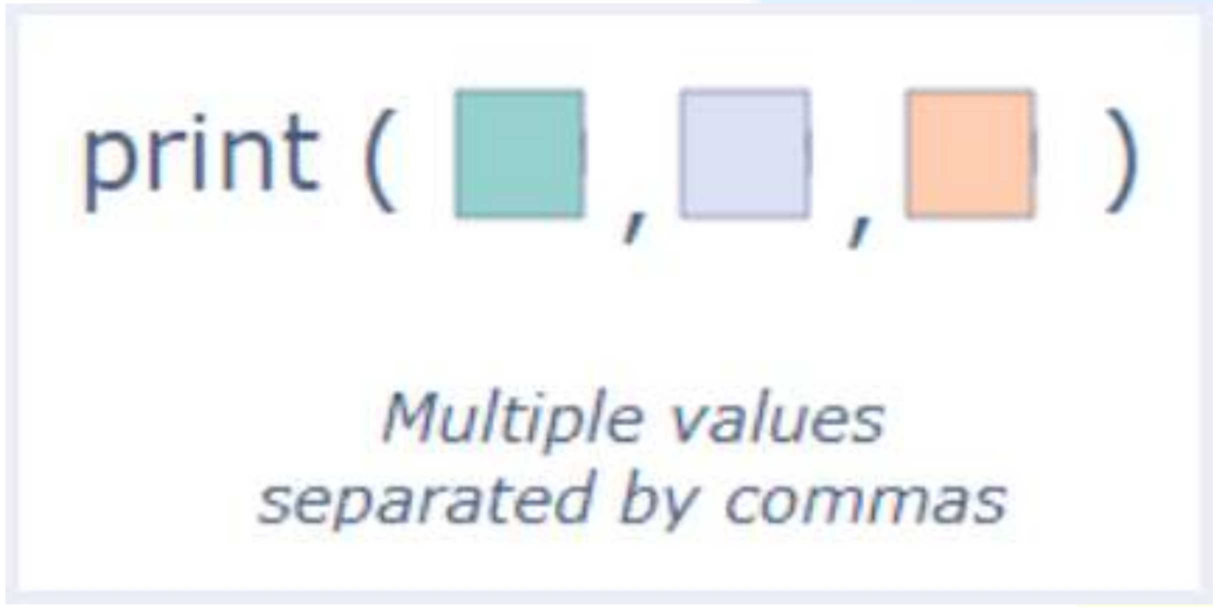
Next, we’ll print a few numbers. Each call to print moves the output to a new line:

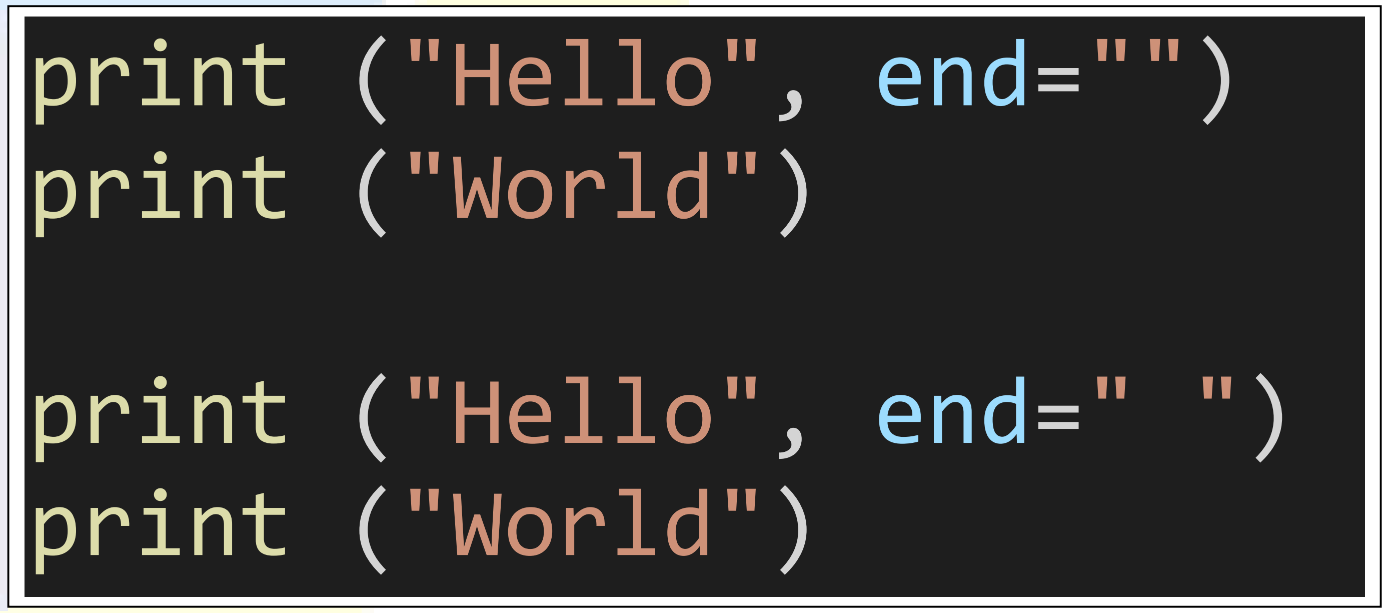
```
print (100)
print (1500)
print (3.142)
```

```
100
1500
3.142
```

Printing Multiple Pieces of Data

Printing multiple things in a single print command; we just must separate them using commas if we want multiple print statements to print in the same line, we use "end" parameter.





```
print ("Hello", end="")
print ("World")

print ("Hello", end=" ")
print ("World")
```

Comments

Comments are pieces of text used to describe what is happening in the code. A comment can be written using the “#” character:

```
print ("Hello World") #This line prints Hello World

# This comment line cannot be displayed in the output

# For multi-line comments, we have to
# Add the hashtag (#) symbol
# at the beginning of each line
```

An alternative to these multi-line comments (line 3-7) are docstrings. They are encased in triple quotes,"""

```
print (150) #This line prints 150

"""
Docstrings are so cool
for writing a comment that
is too long depending on what
the comment is describing in
your codes
"""
```

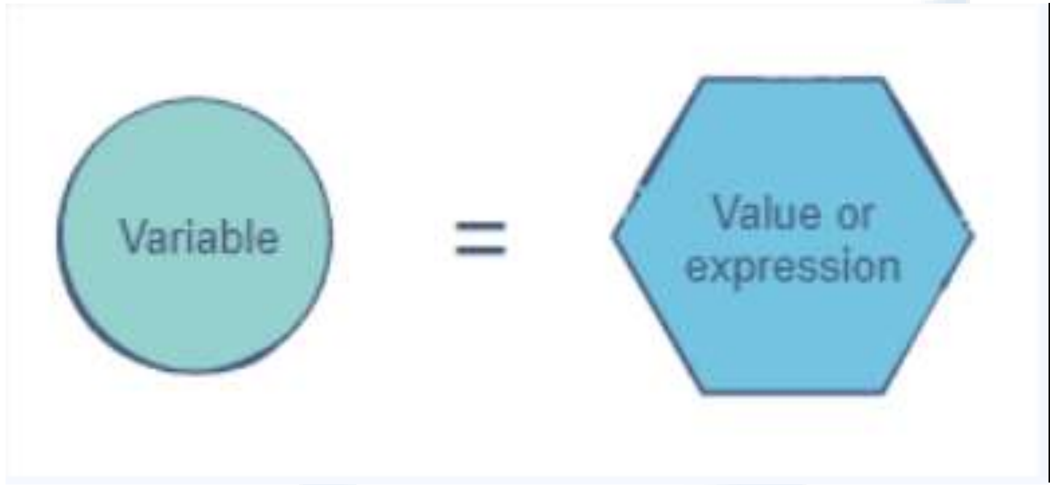
Variables

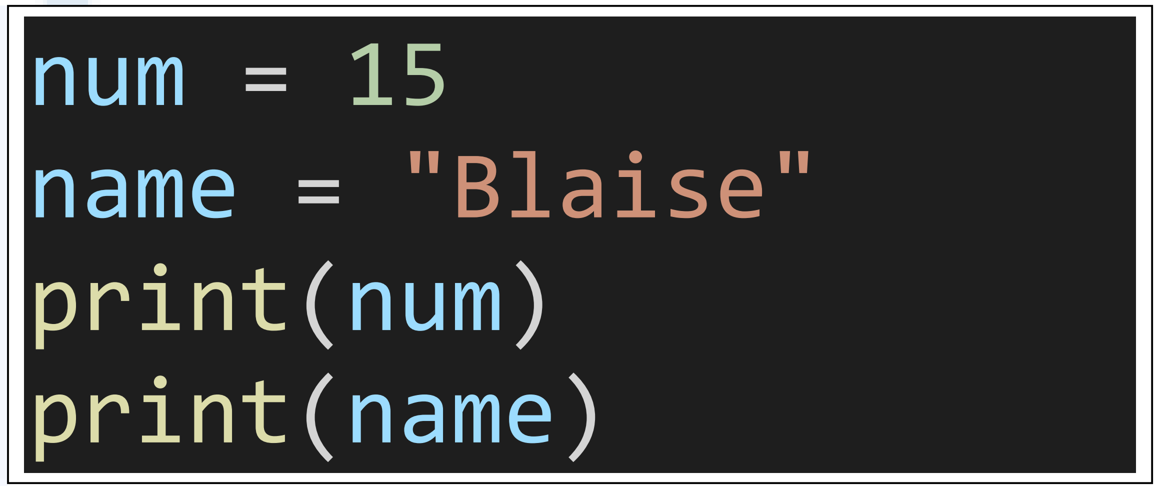
Variables are containers for storing data values. Python variables are mutable. In python, some variables can change their value after creation while some cannot. Therefore, if a variable can change its value it is **mutable** in nature.

Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it. The simplest way to assign a value to a variable is through the "=" operator.





```
num = 15
name = "Blaise"
print(num)
print(name)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Variables Naming Rules

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Examples:

```
myvar = "Peter"
my_var = "John"
_my_var = "Gabriel"
myVar = "Ange"
MYVAR = "Yvonne"
myvar2 = "Alice"
```

Assign Multiple Values

Python allows you to assign values to multiple variables in one line:

```
name, regno, age = "Yvonne", "20RP0001", 22
print (name)
print (regno)
print (age)
```

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

```
fruit = urubuto = "Orange"

print (fruit)
print (urubuto)
```

Types of variables

Global Variables

The variables that are declared outside the scope of a function are defined as global variables in python. Alternatively, the python global variables are considered declared in a global scope. As a result, a user can access a global variable inside or outside the function.

The following example illustrates how a global variable can be created in python:

```
#Create global variable x
x = "global"

#Call x inside a function block
def foo():
    print("x inside:", x)

#Call the foo() function
foo()
#Call x variable out of the function block
print("x outside:", x)
```

Output will be:

```
x inside: global
x outside: global
```

In this particular example, x was created as a global variable. Subsequently, a function foo() was defined to print x. Lastly, we could print the value of the global variable x by calling f00().

In the same example, however, if one wished to change the value of the global variable inside the function, they would face certain complications:

```
#Create global variable x
x = "global"
#Call x inside a function block
def foo():
    x = x * 2
    print(x)
#Call the foo() function
foo()
```

Now, the output will look like this:

```
UnboundLocalError: cannot access local variable 'x' where
it is not associated with a value
```

We get this error because, within the function f00 (), x hasn't been explicitly declared as a global variable. As a result, Python treats it as a local variable and restricts all attempts to update its value. To avoid situations like these, one needs to use the global keyword.

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the global keyword.

Example

```
def myfunc():
  global x
  x = "fantastic"
myfunc()
print("Python is " + x)
```

Also, use the global keyword if you want to change a global variable inside a function.

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = "awesome"
def myfunc():
  global x
  x = "fantastic"
myfunc()
print("Python is " + x)
```

Local Variables

Local variables in python are those variables that are declared inside the function. Alternatively, they are said to defined within a local scope. A user can only access a local variable inside the function but never outside it. Let us take the help of a simple example to understand this:

```
def sum(x,y):
    sum = x + y
    return sum
print(sum(5, 10))
```

The output of this code block will be **15.**

In this case, the local variables declared are x and y. They can only be reached within the scope of the function in which they are declared i.e. sum(). Outside the scope of sum(), they don't exist, and should one attempt to use them, they'll encounter NameError. To gain a better understanding of this error, let us consider the following example:

```
def foo():
    y = "local"
foo()
print(y)
```

The output of this snippet will be:

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

Here, y has been declared as a global variable. Since we tried to access it outside the scope of the function in which it is defined i.e. foo(), we encountered NameError. Thus it is futile to attempt to access a local variable in a global scope because their functionality will always remain confined to a local scope.

Python Data Types

Built-in Data Types

In programming, data type is an important concept.

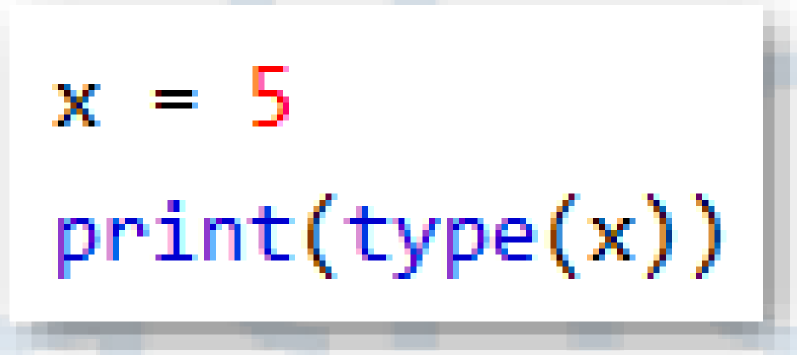
Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	**str**
Numeric Types:	**int, float, complex**
Sequence Types:	**list, tuple, range**
Mapping Type:	**dict**
Set Types:	**set, frozenset**
Boolean Type:	**bool**
Binary Types:	**bytes, bytearray, memoryview**
None Type:	**NoneType**

Getting the Data Type

You can get the data type of any object by using the type() function:



```
x = 5
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

```python
a = "Hello World"                               #str
b = 20                                          #int
c = 20.5                                        #float
d = 1j                                          #complex
e = ["apple", "banana", "cherry"]               #list
f = ("apple", "banana", "cherry")               #tuple
g = range(6)                                    #range
h = {"name" : "John", "age" : 36}               #dict
i = {"apple", "banana", "cherry"}               #set
j = frozenset({"apple", "banana", "cherry"})    #frozenset
k = True                                        #bool
l = b"Hello"                                    #bytes
m = bytearray(5)                                #bytearray
n = memoryview(bytes(5))                        #memoryview
o = None                                        #NoneType
```

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
x = 35e3
y = 12E4
z = -87.7e100
print(type(x))
print(type(y))
print(type(z))
```

Complex

Complex numbers are written with a "j" as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j
print(type(x))
print(type(y))
print(type(z))
```

Complex numbers are made up of a **real** and an **imaginary** part.

Just like the **print()** statement is used to print values**, complex()**is used to create complex numbers.

complex (real, imaginary)

Example:

```
print(complex(10, 20))       #The complex number (10 + 20j)
print(complex(2.5, -18.2))   #The complex number (2.5 - 18.2j)

complex_1 = complex(0, 2)    #The complex number (2j)
complex_2 = complex(2, 0)    #The complex number (2 + 0j)

print(complex_1)
print(complex_2)
```

The output will be

```
(10+20j)
(2.5-18.2j)
2j
(2+0j)
```

Python Casting

If you want to specify the data type of a variable, this can be done with casting.

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
x = int(10.5)     # y will be 10
y = str(10)       # x will be '10'
z = float("10")   # z will be 10.0
```

```

print(type(x))
print(type(y))
print(type(z))
```

Python Booleans

The Boolean (also known as bool) data type allows us to choose between two values: true and false

A Boolean is used to determine whether the logic of an expression or a comparison is correct.

It plays a huge role in data comparisons.

In fact, there are not many values that evaluate to **False**, except empty values, such as **()**, **[]**, **{}**, **""**, the number **0**, and the value **None**. And of course, the value **False** evaluates to **False**.

```
print(True)                                         #True
f_bool = False
print(f_bool)                                       #False
print(10 == 9)                                      #False
print(bool("Hello"))                                #True
print(bool(15))                                     #True
print(bool(["apple", "cherry", "banana"]))          #True
print(bool(None))                                   #False
print(bool(0))                                      #False
print(bool(""))                                     #False
print(bool(()))                                     #False
print(bool([]))                                     #False
print(bool({}))                                     #False
```

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

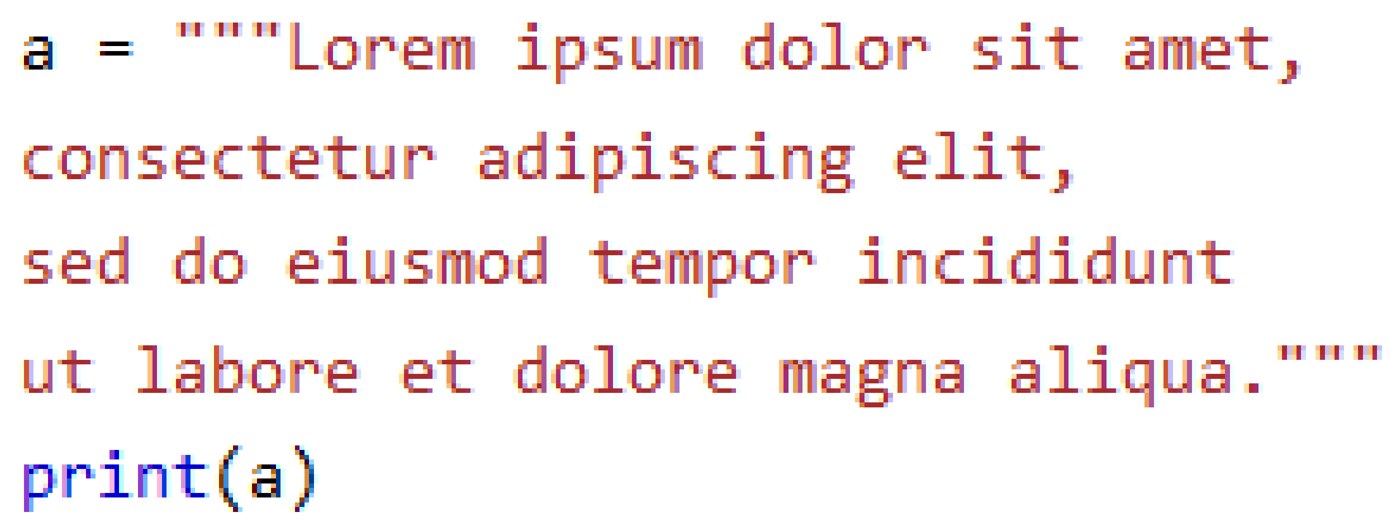
You can display a string literal with the print() function:

```
print("Hello")
print('Hello')
```

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

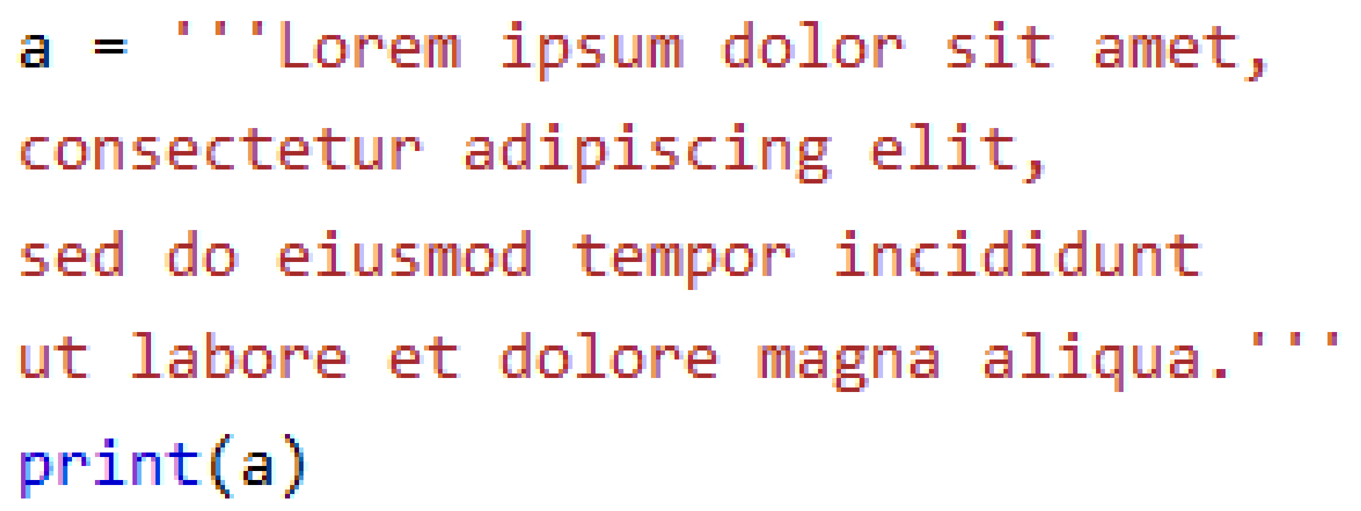
```
a = "Hello"
print(a)
```

You can assign a multiline string to a variable by using three quotes:



```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:



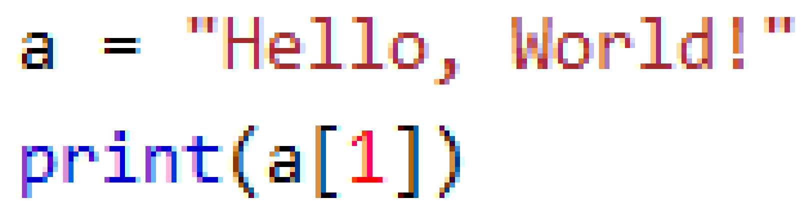
```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

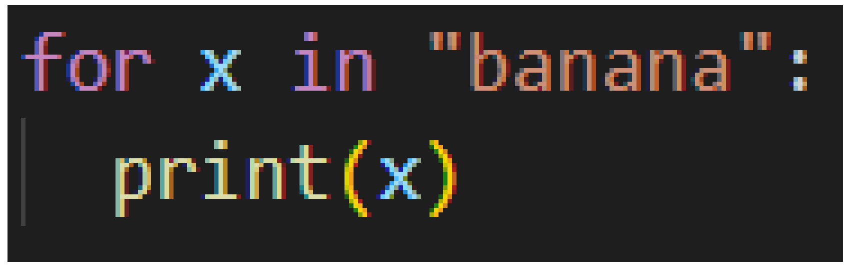
Example: Get the character at position 1 (remember that the first character has the position 0):



```
a = "Hello, World!"
print(a[1])
```

Since strings are arrays, we can loop through the characters in a string, with a for loop.

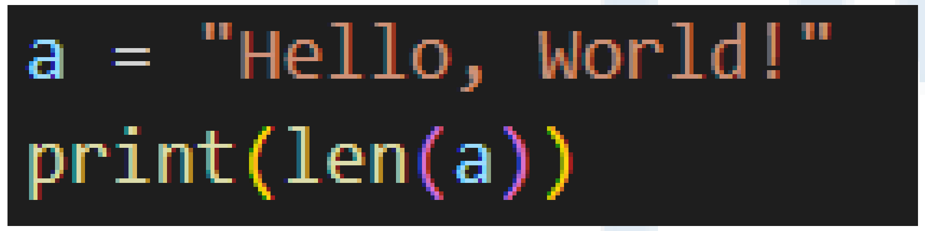
Example - Loop through the letters in the word "banana":



```
for x in "banana":
  print(x)
```

To get the length of a string, use the **len()** function.

Example - The len() function returns the length of a string:



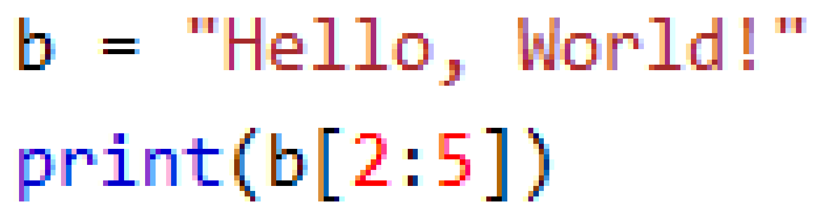
```
a = "Hello, World!"
print(len(a))
```

Slicing a String

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

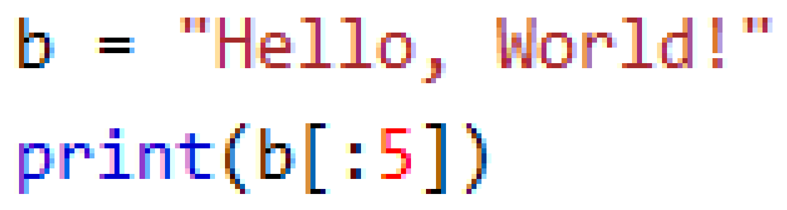
Example - Get the characters from position 2 to position 5 (not included):



```
b = "Hello, World!"
print(b[2:5])
```

- Output will be **“llo”**

Get the characters from the start to position 5 (not included):



```
b = "Hello, World!"
print(b[:5])
```

- Output will be **“Hello”**

Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"
print(b[2:])
```

- Output will be **“llo, World!”**

Negative Indexing

Use negative indexes to start the slice from the end of the string:

Example:

Get the characters:

From: "o" in "World!" (position -5)

To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

- Output will be **“orl”**

Modifying String

The **upper()** method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

The **lower()** method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

The **strip()** method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

The **replace()** method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

The **split()** method splits the string into substrings if it finds instances of the separator:

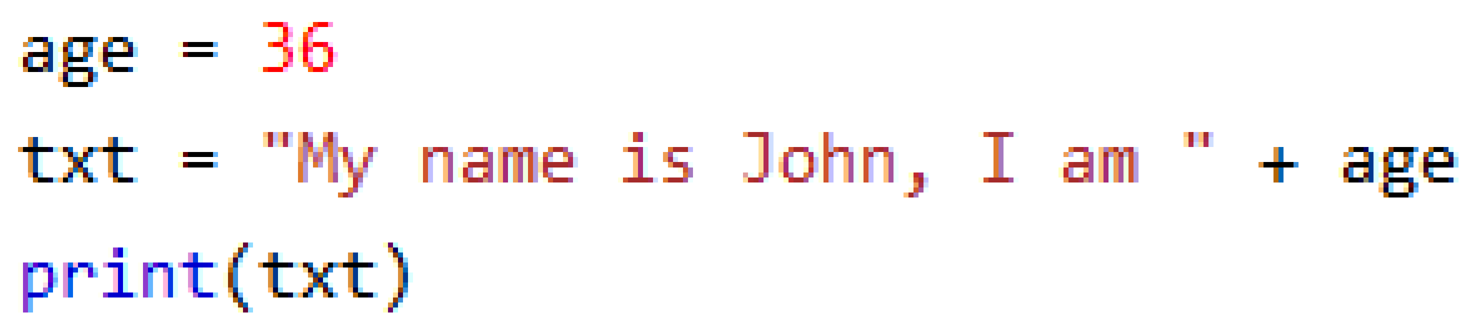
```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

String Concatenation

Merge variable a with variable b into variable c:

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

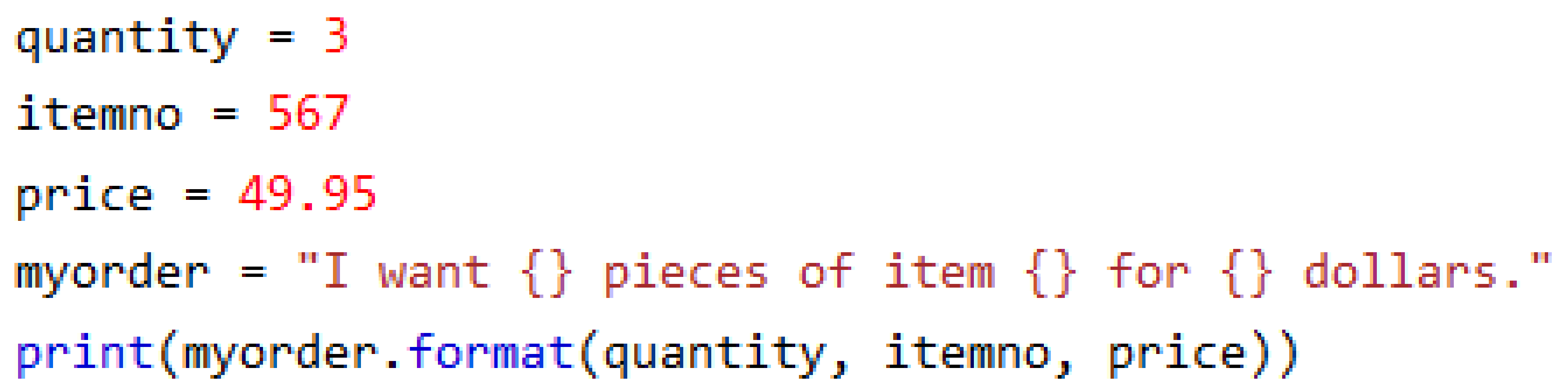
As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:



```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

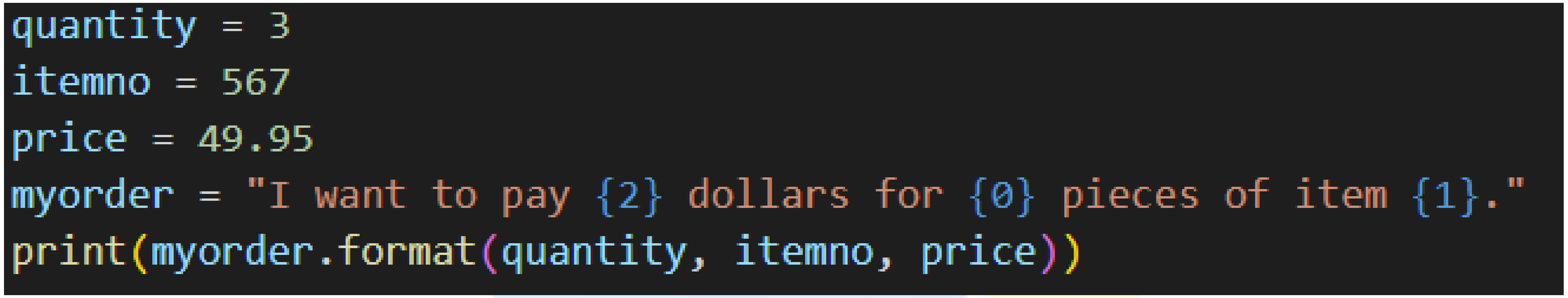
But we can combine strings and numbers by using the **format()** method!

The **format()** method takes the passed arguments, formats them, and places them in the string where the placeholders **{}** are:



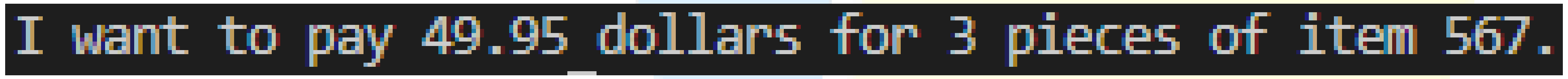
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use index numbers **{0}** to be sure the arguments are placed in the correct placeholders:



```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

The output is:



```
I want to pay 49.95 dollars for 3 pieces of item 567.
```

Escape Characters

Code	**Result**
\’	Single Quote
\\\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace

\f	Form Feed
\ooo	Octal value
\xhh	Hex value

String Methods

Method	**Description**
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value

expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case

isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found

rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case

translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning