Python for Data Analysis: Methods & Tools

Python for everyday people

Written by Felipe Dominguez - Professor Adjunct
Hult International Business School

Chapter 01 - Python Fundamentals

Help, Comments, Printing, Dynamic Strings, User Input, and more!

1. Ask for HELP!

1.1. Help will always be given in Python to those who ask for it.

Are you struggling to understand a new piece of code? Are there functions present that you are unfamiliar with? Don't worry, Python has a built-in function called **help()** that can provide information about any function. To use it, simply call **help()** on the function in question. But before you do that, let's make sure you understand what functions are

1.2. Understanding the print() function.

A function is a block of code that can be stored in a virtual or local Python environment and is executed when called upon. You can also pass parameters or arguments into a function, and it may return a result. Actually, one of the most commonly used functions in Python is the print() function. You can use the help() function to learn more about print() by passing it as an argument. Simply call **help(print)** to get a description of the function.

Don't hesitate to ask for help when you need it!

In [28]:
```
# Code 1.2.1.
# Calling help function over print function
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

The **help** function usually return the same format for each function. However, it will depend on the "DocString" documentation written by the creator. Let's dive into the print function and understand how it works.

Let's enumerate each line from the previous output.

#	Description
1	Help on built-in function print in module builtins:
2	
3	print(...)
4	print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
5	
6	Prints the values to a stream, or to sys.stdout by default.
7	Optional keyword arguments:
8	file: a file-like object (stream); defaults to the current sys.stdout.
9	sep: string inserted between values, default a space.
10	end: string appended after the last value, default a newline.
11	flush: whether to forcibly flush the stream.

Let's understand what each row means.

#	Explanation
1	Convey the message the print function is a **built-in function**
2	
3	Name of the function looking at
4	State the arguments for this function
5	
6	Describe what the function does
7	Describe the arguments of the function
8	file: a file-like object (stream); defaults to the current sys.stdout.
9	Define the character between string values. Try sep = "-"
10	Define character after the last value. Try end = "..."
11	Define the stream/buffer of the output

2. Comment, Comments Commented, Commenting

Now that you know how to ask for help when you need it, it's important to also learn how to effectively communicate your own work to others. As a data scientist, data analyst, data engineer, or any other role, you will likely work with colleagues and need to explain your own work to them. Adding comments to your code can help others understand your logic, facilitate communication and collaboration, and provide insight into your thought process.

To add a comment to your code, use the **#** character.

Try the following lines of code to see how comments work in action.

In [3]:
```
# Code 2.a.
# This is a comment. There is no output.
```

In [30]:
```
# Code 2.b.
This is not a comment. You will get an error.
```

```
  Input In [30]
    This is not a comment. You will get an error.
                   ^
SyntaxError: invalid syntax
```

2.1. Talk to humans, work with machines

Try reading the code 2.1.1.

In [1]:
```
# Code 2.2.1. #
import numpy as np

while True:
    try:
        budget = int(
            input(prompt="Hello there, what is your weekly grocery budget?"))
        break
    except:
        print("Please enter a number.")

g_cost = {"apples": 1, "milk": 3, "cheese": 5}


def groceries(g_cost, budget, m_list):
    """
    Function to solve a linear equation with 3 unknown variables

    Parameters
    ----------

    g_cost: dictionary with the cost of each grocery item
    budget: Budget limit
```

```
    m_list: list with the linear equations
    """

    A = np.array(m_list)
    inv_A = np.linalg.inv(A)
    B = np.array([budget, 0, 0])
    X = np.linalg.inv(A).dot(B)
    return X


m_list = [[g_cost["apples"], g_cost["milk"], g_cost["cheese"]],
          [2, -1, 0],
          [0, 1, -1]]
X = groceries(g_cost, budget, m_list)

print(f"""
    We can buy {round(X[0])} apples, {round(X[1])} liters of milk,
    and {round(X[2])} slices of cheese with {budget} dollars.
""")
```

```
Hello there, what is your weekly grocery budget?200

    We can buy 12 apples, 24 liters of milk,
    and 24 slices of cheese with 200 dollars.
```

2.2. Let's add comments!

Code 2.1.1 might seem intimidating and hard to understand at first glance. While you could go through each line and function using the help() function as you learned in section 1, this can be time-consuming. This is where comments become especially helpful.

Comments allow you to explain the logic and purpose of your code, making it easier for others to understand.

Let's take a look at code 2.2.2 now.

In [8]:
```
# Code 2.2.2. #

# Importing packages
import numpy as np

# Ask the user for the budget. Convert user input into a number
# Control potential user errors with while and try/except functions
while True:
    try:
        budget = int(
            input(prompt="Hello there, what is your weekly grocery budget?"))
        break
    except:
        print("Please enter a number.")


# Dictionary to define the cost of each grocery element (apples, milk, and chee
g_cost = {"apples": 1, "milk": 3, "cheese": 5}
```

```
# Creating groceries function
def groceries(g_cost, budget, m_list):
    """
    Function to solve a linear equation with 3 unknown variables

    Parameters
    ----------

    g_cost: dictionary with the cost of each grocery item
    budget: Budget limit
    m_list: list with the linear equations
    """

    # Transform m_list into an numpy array (matrix)
    A = np.array(m_list)

    # Using linear algebra to invert A matrix
    inv_A = np.linalg.inv(A)

    # Create B matrix with the vector of results
    B = np.array([budget, 0, 0])

    # Find the solution, using the algebra formula A^-1 * B = X
    X = np.linalg.inv(A).dot(B)

    # Return the value of each unknown variable (number of apple, milk, and che
    return X

# Assumptions:
    # We can't spend more money than the budget
    # Customer would prefer to buy 2 liters of milk rather than 1 apple
    # Customer wants to buy the same number of liter of milk and slices of chee


# Define a matrix equation based on assumptions
    # Assumption 1: 1*apple + 3*milk + 5*cheese = budget
    # Assumption 2: 2*apple - 1*milk + 0*cheese = 0
    # Assumption 3: 0*apple + 1*milk - 1*cheese = 0
m_list = [[g_cost["apples"], g_cost["milk"], g_cost["cheese"]],
          [2, -1, 0],
          [0, 1, -1]]
# Create a variable X that calls the function groceries
X = groceries(g_cost, budget, m_list)

# Print and add context to the solution
print(f"""
    We can buy {round(X[0])} apples, {round(X[1])} liters of milk,
    and {round(X[2])} slices of cheese with {budget} dollars.
""")
```

```
Hello there, what is your weekly grocery budget?200

    We can buy 12 apples, 24 liters of milk,
    and 24 slices of cheese with 200 dollars.
```

Can you see the difference between code 2.2.1. and 2.1.2.? Even if there are still many concepts that you have yet to learn, comments can help you understand the reasoning

behind a program. When writing code, always consider how someone who is unfamiliar with your work would be able to comprehend it. Use comments to clarify your thought process and make your code more accessible to others.

3. Let's Print!

3.1. Hello, World!

Let's put our knowledge to use and create our first program! In the previous section, you learned how to ask for **help()** with code, and you also looked at the **print()** function. Now, let's use these skills to create a simple program that says "Hello, World!" to everyone.

In [31]:
```
# Code 3.2.1.
print("Hello, World!")
```

```
Hello, World!
```

3.2. Hello, World! in a different way

As expected the program output a string with the text "Hello, World!". Congrats! You just run your first piece of code! Let's evaluate variations of the print function.

In [57]:
```
# Code 3.2.2.

# print with more than 1 string value. Concat two string values
print("Hello,", "World!")

# print with sep = ";". It will connect two or more string values with ";"
print('Hello', 'World!', sep=";")

# print with end = "...". At the end of the sentence will add "..."
print("Hello, World!", end="...")

# print in multiple lines
print("""
Hello,
World!""")
```

```
Hello, World!
Hello;World!
Hello, World!...
Hello,
World!
```

As you can see, using different arguments with the print( ) function allows you to **modify the output of the program**. Whenever you encounter a new function, remember to use the help( ) function to learn more about it, examine the available arguments, and determine the best way to meet your needs

3.3. Print Variables!

Let's talk about variables in Python.
In Python, a variable is a reserved memory location used to store a value. Variables can be given any name (except for Python reserved words [4.1.]), they can automatically adapt in size, store any type of data and exist only when they are told so. However, it is important to **choose descriptive and intuitive names for your variables** to avoid confusion and ensure that you understand what is being stored. You can't have two variables with the same name (unless you want to overwrite it).

Variable names can contain uppercase and lowercase letters, underscores, and numbers (except for the first character). In this class you will will follow the PEP 8 Naming style.

Now, let's update our "Hello, World!" program to use variables instead of hard-coded values. See code 3.3.1 for an example.

In [161…
```
# Code 3.3.1.

# Declare variables part_1 and part_2
part_1 = "Hello,"
part_2 = "World!"

print(part_1, part_2)
```

```
Hello, World!
```

As demonstrated in code 3.3.1, you can use variables to create a print( ) statement by simply separating the variables with a comma. Code 3.3.2 shows a more advanced example of using variables with print( ), but the general concept remains the same. By using variables, you can easily customize the output of our program and make it more flexible.

In [1]:
```
# Code 3.3.2.

# Let's work with a more difficult print and variables code.

# Declare variables cars, space_per_car, drivers, and passengers.
cars = 200
space_per_car = 4
drivers = 15
passengers = 80

# Declare variables based in previous variables.
# How many cars are without a driver?
cars_not_driven = cars - drivers

# How many cars have a driver?
cars_driven = drivers

# What is the total capacity to transport passengers?
total_pass_capacity = cars * space_per_car
```

```
# What is the maximum number of passenger possible to transport?
max_pass_transport = cars_driven * space_per_car

# Let's print stamtents with variables!
print("There are", cars, "cars available.")
print("There are only", drivers, "drivers available.")
print("We can drive", drivers, "cars out of the", cars, "cars available today.'
print("At full capacity, we can transport",
      total_pass_capacity, "passengers per day")
print("We can transport", max_pass_transport, "passengers today")
print("")

# Try changing the original variables, and see how all the statements change!
```

```
There are 200 cars available.
There are only 15 drivers available.
We can drive 15 cars out of the 200 cars available today.
At full capacity, we can transport 800 passengers per day
We can transport 60 passengers today
```

3.4. Printing vs Output!

There is a subtle distinction between using print( ) and displaying cell output in Jupyter Notebook. While code 3.4.1 may seem to have the same result as code 3.2.1, adding an additional line of code, such as "2 + 2", after "Hello, World!" will result in a different output.

By default, Jupyter will always display the output of the last line of code, regardless of whether the print() function is used. Therefore, any lines preceding the last one will not be displayed unless the print() function is called. This is why it is important to use print() statements to ensure that all desired output is shown

Always remember, **output is NOT the same as print.**

In [53]:
```
# Code 3.4.1.
# Output
'Hello, World!'
```

Out[53]:
```
'Hello, World!'
```

In [55]:
```
# Code 3.4.2.
# Multiple lines Output
'Hello, World!'

2 + 2
```

Out[55]:
```
4
```

In [54]:
```
# Code 3.4.3.
# print and output

print("Hello, World!")
```

```
2 + 2
```

```
Hello, World!
```

Out[54]:
```
4
```

4. Python Keywords and Escape Sequences.

4.1. Python reserved keywords

You may have noticed that some words in code 2.a. are highlighted in different colors. These words, such as **"is"** and **"not"**, are special to Python and are used for specific tasks. They are highlighted in green, which is the way Python has to tell you that those words are special. As a result, they cannot be used as variable names.

Any word that is not on the list of Python keywords can be used as a variable. Refer to table 4.1 for a list of common Python keywords

Value	Operator	Conditional	Iteration	Structural	Return	Import	Exemptio
True	and	if	for	def	return	import	try
False	or	elif	while	class	yield	from	Exception
None	not	else	break	with	-	as	raise
-	in	-	continue	pass	-	-	finally
-	is	-	-	lambda	-	-	assert

Table 4.1: Python data types.

4.2. Wrappers & Escape Sequences

4.2.1. Wrappers

Wrappers are syntax elements that help you enclose another piece of syntax. You have already used wrappers when you wrote the print() function:

```
print('Hello world!')
```

In this example, the parentheses and single quotes are both types of wrappers. There are three types of wrappers for quotation (strings) in Python:

```
a. ""
b. ''
c. """"""""
```

As shown in code 3.2.2, you can use a. and b. interchangeably. However, c. allows you to write statements across multiple lines, as shown in code 4.2.1 (a).

What do you do when you need to include a quotation mark within a string, as in code 4.2.1 (b) and (d)? Using the same type of quotation wrapper will result in an error, as Python will interpret the string as ending at the quotation mark. To resolve this issue, you can use both types of quotation marks simultaneously and keep them consistent, as shown in code 4.2.1 (c). Refer to table 4.2.1 for a list of **common wrapper types**.

4.2.2. Escape Sequences

Since some characters have specific functions in Python, it is important to be careful when writing code. There may be times when you want to remove the special meaning of certain words and use them as a string instead. Python provides a built-in character, the backslash **(\)** that allows you to toggle the special meaning of certain characters.

For example, if you try to run code 4.2.1 (d), you will encounter an error. However, by examining code 4.2.1 (e), you can see that a backslash has been added before the double quotation mark. This tells Python to treat the quotation mark as a regular character, rather than a string wrapper. Refer to table 4.2.1 for a list of common escape sequences.

In [56]:
```
# Code 4.2.1.(a)
# printing with """""""

print("""
This is a string
that can be written in
as many lines as we
would like to have
""")
```

```
This is a string
that can be written in
as many lines as we
would like to have
```

In [ ]:
```
# Code 4.2.1.(b)

# Using only single quote.

print('We don't talk about Bruno)
```

In [51]:
```
# Code 4.2.1.(c)

# Fixing Code 4.2.1.(b)
# Interchange Single & Double Quotes

print("We don't talk about Bruno")
```

```
We don't talk about Bruno
```

In [ ]:
```
# Code 4.2.1.(d)

# Using only double quotes
print("Milton Friedman said: "There is no such thing as a free lunch" ")
```

In [60]:
```
# Code 4.2.1.(e)

# Fixing Code 4.2.1.(d)
# Using \ as a escape secuence

print("Milton Friedman said: \"There is no such thing as a free lunch\"")
```

```
Milton Friedman said: "There is no such thing as a free lunch"
```

Syntax	Description
print()	outputs an argument to the console
' '	wrapper to print strings
" "	wrapper to print strings
""" """	wrapper to print strings on multiple lines
f" {var} "	dynamically calls variables/objects into print strings
f""" {var} """	dynamically calls variables/objects into print strings on multiple lines

*Table 4.2.1: Common **print()** wrappers.*

Escape	Description
\\	escapes the special meaning of the backslash (/)
\'	escapes the special meaning of the single quote wrapper
\"	escapes the special meaning of the double quote wrapper
\b	activate the special meaning of 'b' remove previous character
\v	activates the special meaning of 'v' add a vertical tab (space)
\n	activates the special meaning of 'n' (new line)
\t	activates the special meaning of 't' (horizonal tab)

Table 4.2.2.: Common escape sequences.

5. Dynamic Strings

Code 3.2.2 illustrates how to use variables in print statements to create **dynamic strings** or **f-strings**. It's important to keep in mind that every new variable you create will use memory, which can slow down your working environment. To minimize the number of variables in your code, you can use dynamic strings to include variables or calculations within a string by placing the letter "f" before the string in the print statement. Any variables or calculations within curly brackets {} in a dynamic string will be treated as code by Python. For example, let's use dynamic strings to find out how many days are left until Christmas.

In [1]:
```
# Code 5.1.1.
# How many days are left to Christmas?

# You could manually calculate the days between today and Christmas (1/9/23 to
days_to_christmas = 349

# print days to christmas
print(f"Only {days_to_christmas} days to Christmas!")
```

```
Only 349 days to Christmas!
```

Code 5.1.1 gives you the number of days until Christmas using a variable. However, if you want this information to stay current, you will need to update the number of days every day.

Code 5.1.2 demonstrates a dynamic way to obtain today's date by importing the datetime package.

In [127…
```
# Code 5.1.2.
# How many days are left to Christmas?

# import datetime package
import datetime

# get today's date with the function now of datetime (method chaining)
today = str(datetime.datetime.now().strftime("%m/%d/%Y"))

# print today's date
print(f"Today is {today}")
```

```
Today is 11/13/2022
```

You are almost there! You know how to dynamically get today's date. Now let's get the number of days between Christmas and today.
Look at Code 5.1.3.

In [4]:
```
# Code 5.1.3.
# How long to be Christmas?

# import datetime package
import datetime

# get today's date with the function now of datetime (method chaining)
today = str(datetime.datetime.now().strftime("%m/%d/%Y"))

# Format Christmas date
christmas_day = datetime.datetime.strptime("12/24/2023", "%m/%d/%Y")

# Calculate days between today and Christmas. Return days.
days_to_christmas = (christmas_day - datetime.datetime.now()).days

# print the final result
print(
    f"""
    ------------------------------------------------------------
    Today is {today}, that means there's
        Only {str(days_to_christmas)} days to Christmas!
    \t\t\t Are you ready?!
```

```
    ------------------------------------------------------------
    """)
```

```
    ------------------------------------------------------------
    Today is 12/22/2022, that means there's
        Only 366 days to Christmas!
                        Are you ready?!
    ------------------------------------------------------------
```

6. Ask the user! Use their INPUT

6.1. Don't hesitate, just ask!

It is common to need to interact with the end user of a program. For example, you may need to prompt the user to select an option, provide input, or trigger the next part of the program. The built-in function **input** allows you to gather information from the user and customize their experience. This technique is known as "prompting the user."

Let's learn more about it.

6.2. Prompting Users

As the name suggests, the **input function** prompts the user for information. When you run the **input()** function, a small rectangle will appear in the user interface, where the user can enter anything. Python will automatically convert any input into a string. It's important to keep in mind that you may need to transform the input into a different data type, depending on your needs. As you can imagine, it would be useless to ask for input if you don't save it for the future. Therefore, you need to store it in a variable. Code 6.2.2 demonstrates how to do this.

Note that you cannot run a different cell while the input function is still running. The Python kernel will continue to run the cell until you enter something into the prompt (or click "enter") or force the cell to stop. If you move too quickly through your code and do not see an output, check for a star **(*)** in one of the previous cells, as it could indicate that there is an open user prompt. Make sure to understand why the cell has not stopped running.

In [144…
```
# Code 6.2.1.
# Ask the user for it's name
input()
```

```
Felipe
```

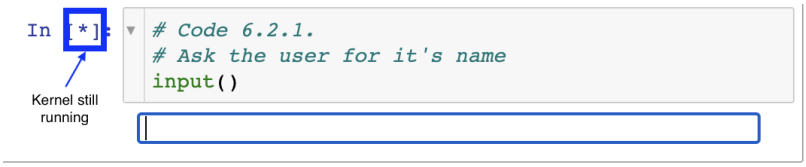
Out[144]:
```
'Felipe'
```

In [142…
```
# Code 6.2.2.
# Store user name into user_name variable
```

```
user_name = input()

print(f"The name of the user is {user_name}")
```

```
Felipe
The name of the user is Felipe
```



6.3. Ask Input, Be clear, Save it.

As mentioned in Section 2.1, it's important to **"talk to humans, work with machines."** On code 6.2.2. you ask the user for input, but... about what. You can't assume the user will intuitively understand everything. It's important to provide context or information about what you are asking for. Thankfully, you can provide this to users. The **input()** function has an optional argument called **prompt** which allows you to describe the purpose of the input to the user. Be as clear and concise as possible, as the end user may not be familiar with the program. Remember to provide enough context to help the user understand what they are being asked to do.

In [159…

```
# Code 6.3.1.

# Import datetime
import datetime

# Asking and saving user_name
user_name = input(prompt="Hi! Welcome to Python! What's your name? ")

# Asking and saving user age
user_age = input(prompt=f"""It's great to meet you {user_name}.
{user_name}, tell me, how old are you? """)

# calculating year of birth

# Declare variable with today's year.
today_year = datetime.date.today().year

# calculate year birht by substracting today's year with age.
user_year_birth = today_year - int(user_age)

print(f"""
{"-" * 50}
Wow! You were born in {user_year_birth}.
Did you know I was borned in 1991.
This means we have {abs(1991 - user_year_birth)} years of difference!
{"-" * 50}""")
```

```
Hi! Welcome to Python! What's your name? f
It's great to meet you f.
f, tell me, how old are you? 29
Wow! You were born in 1993.
Did you know I was borned in 1991.
This means we have 2 years of difference!
```

In Code 6.3.1, you demonstrated how to guide the user to provide the specific information you need and how to use advanced techniques like converting a variable to an integer or using the abs() function. These concepts will be covered in more detail in the next chapter.

For now, you should feel proud of what you have learned so far, including how to:

- Ask python for help!
- Comment and make your code easy-to-read
- Use print statements
- Understand Python keywords and escape sequences.
- Create dynamic strings.
- Prompt the user for input.