Input &
Working with strings

003_Python Evening Class

Working with strings

Reading Input From the Keyboard

Programs often need to get data from the user, usually by way of input from the keyboard. The simplest way to accomplish this in Python is with input().

```
input([<prompt>])
```

Reads a line of input from the keyboard.

input() **pauses** the program, to allow the user to type in a line of input from the keyboard. Once the user presses the Enter key, all characters typed are read and returned as a **string.**

```
user_name= input("Enter Name :")
print("Hello " + user_name)

user_age= input("Enter Age :")
print(type(user_age))

print(f"I hear you {user_age}")
```

Authors note:
Don't forget about

```
print(help("input"))
```

```
#Why is this erroneous?
user_age  = input("Enter Age :")
user_age += user_age + 1
```

Reading Input From the Keyboard

Note: that the newline generated when the user presses the Enter key isn't included as part of the return string.

If you include the optional <prompt> argument, input() displays it as a prompt to the user before pausing to read input.

input() always returns a string.

As such if you want a numeric type, then you need to convert the string to the appropriate type with the int(), float(), or complex() built-in functions.

```
user_age= int(input("Enter Age :"))
print(type(user_age))

user_age += user_age + 1
```

Dissecting the statement

```
user_name = input( "What's your name :" )
```

`user_name`

As you may have guest from the assignment operator "=" . This will create a variable named "user_name" and store whatever the input function returns.

`input`

this is the function call, note this is a built-in function, so we do not need to tell Python were to find it. You will notice functions usually are followed by opening and closing round brackets ().

`"What's your name :"`

This is the argument, the data that is passed into the function call, note this is optional, but is very useful as you can prompt the user to enter data.

Authors note:
The print function will stop the flow of the programme until the enter key is pressed.

Casting data From the Keyboard

Often you will need to cast the data returned from the input statement into the appropriate data type, as the input statement will always return a string data type.

This can be done in two common ways Consider the two following code snippets

```
number = input("Enter a number:")
number=float(number)
number_squared =number*number
```

```
number = float(input("Enter a number:"))
number_squared =number*number
```

Authors note:
Both are perfectly valid, and function very similarly. There might be a minor argument to be made one is more efficient in regard to accessing memory.

But I would stress the most important element is to be consistent in your approach.

split() *(Method in the string object)*

As we seen the input function will also return a single string,but often when working with strings we need to break them into pieces. For example, commerce separate value (csv) file.

String object provides us with a really helpful method split.
The method split returns a list [] of strings " “

Note: By default, this method will split string on white spaces, But we can specify what to split on by passing in a string parameter.

```
names ="James Joy Tommy Sammy"
list_of_names= names.split()

print(list_of_names)
print(type(list_of_names))
```

```
names ="James-Joy-Tommy-Sammy"
list_of_names= names.split("-")

print(list_of_names)
print(type(list_of_names))
```

Authors note:
It is important to note that the separator is not included in any of the values in the list. And the string separator argument can be of any length greater than zero.

Don't forget about:

```
print(help("str.split"))
```

Get multiple input values

```
num_1,num_2,num_3 ="089 123 1234".split(" ")
print(num_1,num_2,num_3 )
```

For example,
In a single execution of the input() function, we can ask the user there name, age, and phone number then **unpack** & store these three different variables.

```
name,age,phone = input("enter name age phone :")
print(name,age,phone)
```

However, this **unpacking** approach only works if split() returns the same number of elements as variables are provided.

eval (Built in function)

The above function takes a string argument and evaluates it as if it was Python code. This could be very useful if a little dangerous.

This will allow the user to write any statement directly into the program and the Python interpreter will read this as if it's part of the code.

In general, any valid Python expression stored as text in a string s can be evaluated by eval.

The parameters of the eval function must be expressions in quotes, otherwise an error will be reported.

The result of an expression is an object.

Eval will result in an error if, statements are used inside it.

```
print(eval("num=2"))
```

```
sum=("1+1")
print(eval(sum))

user_input=input("Enter a Sum:")
print(eval(user_input))

bool=("True")
print(eval(bool))

print(eval("pow(2,2)"))

num=1
print(eval("2+num"))


var= eval("'Hello'")
#print(type(var), var) #?
```

Authors note:
Don't forget about

```
print(help("eval"))
```

Applying eval() to user input

So, why is the eval function so useful?

When we get input via input function, it is always in the form of a string object, which often must be converted to another type of object, usually an int or float.

Sometimes we want to avoid specifying one type. The eval function can then be of help: we feed the string object from the input to the eval function and let the it interpret the string and convert it to the right object.

```
number = eval(input("Enter a number"))
sum= number + 1
print(type(sum))
```

Command Line arguments

Another way to pass input to the program is Command-line arguments.
Command line arguments are flags given to a program/script at runtime.

In a command-line argument, the input is given to the program through command prompt rather than python script like input() Function. This allows us to give our program different input on the fly without changing the code.

For example, if your program processes data read from a file, then you can pass the name of the file to your program, rather than hard-coding the value in your source code.

Authors note:
A command line interface (CLI) provides a way for a user to interact with a program running in a text-based shell interpreter.
Some examples of shell interpreters are Bash on Linux or Command Prompt on Windows.

The sys module

To access the command line arguments, we first need to import the sys module. Then, with the module imported, we can access the commandline argument (argv) attribute by using the dot operator.

The argv attribute is a list data type. You will notice the first value argv[0] in the list is the name of the program, you will also notice all values in this list are strings.

When entering commandline arguments. Each argument must be separated with a whitespace

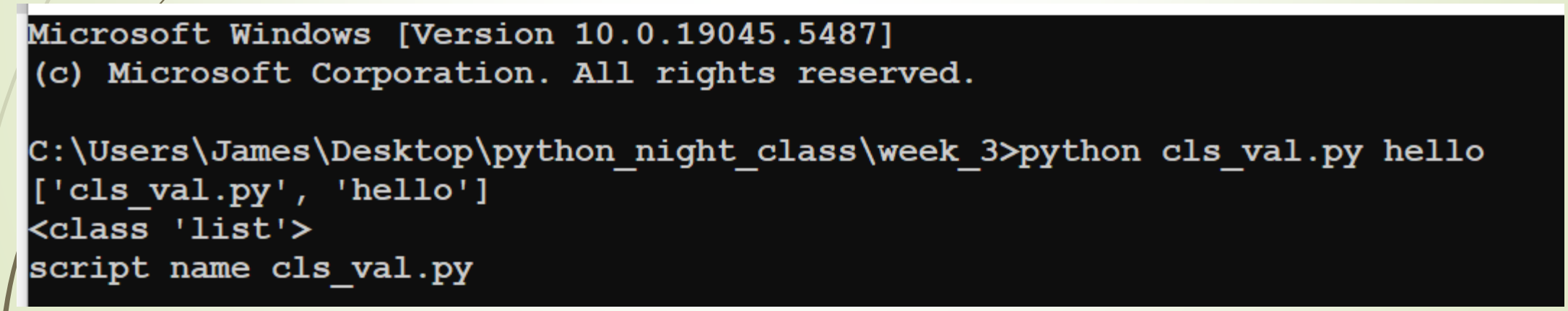
```
import sys


print(sys.argv)
print(type(sys.argv))


print(f"script name {sys.argv[0]}")
```

Passing in commandline arguments

I will show you two ways of passing in commandline arguments.

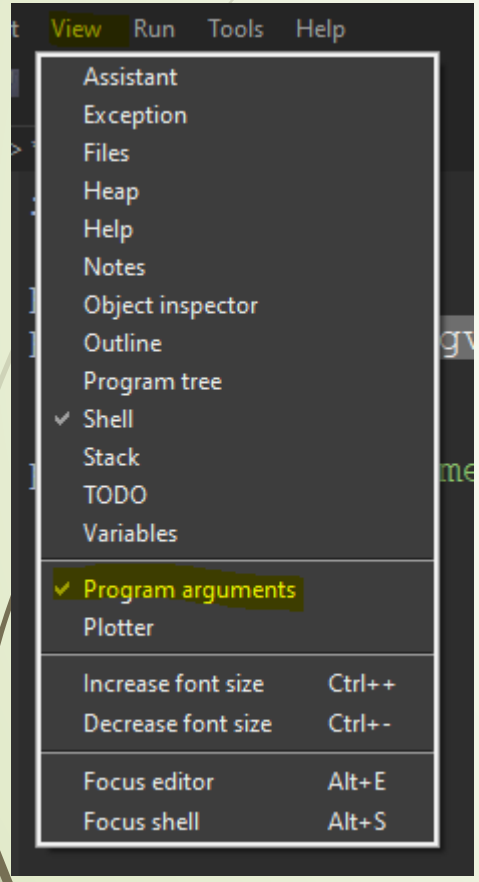
Firstly, in the terminal.
You will need to navigate to the location of the Python file you wish to run,And then enter python , followed by the filename.py and any arguments you wish to pass

When entering commandline arguments. Each argument must be separated with a whitespace

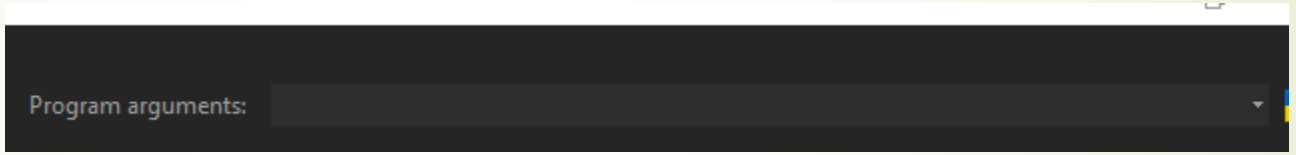


Passing in commandline arguments

Secondly Thonny allows you to passing commandline arguments.
Click view, then from the drop-down menu, make sure program arguments is checked.



In the top right-hand corner, you should see the following dialogue box.



Authors note:
Similarly to input function. All commandline arguments are strings, and as such you may need to cast them into appropriate datatypes.

len *(Built in function)*

The Len will take an iterative (string, list, tuple) and return an integer totalling the number of elements in the iterative

In this second example we simply count the number of command line arguments using the built-in len() *function*.

sys.argv is the list that we have to examine.

NOTE – As mentioned, first argument is always script name and it is also being counted in the number of arguments, so will always have at least a value of one.

```
import sys
print(len(sys.argv))
```

Authors note:
Usually, space is separator between command line arguments. If our command line argument itself contains space, then we should enclose within double quotes (but not single quotes)

Writing Output to Console

We started your Python journey by learning about print(). It helped you write your very own hello world one-liner.

The simplest example of using Python print() requires just a few keystrokes:

```
>>> print()
```

You don’t pass any arguments, but you still need to put empty parentheses at the end, which tell Python to execute the function rather than just refer to it by name.

This will produce an invisible newline character, which in turn will cause a blank line to appear on your screen. You can call print() multiple times like this to add vertical space.
It’s just as if you were hitting Enter on your keyboard in a word processor.

print *(Built in function)*

```
>>> print()
```

We've been using the print function quite a lot, so now let's use it to talk about one or two things.

The print function does not require any arguments, however, even if you're not passing any argument we will want to add the opening, closing round brackets, so Python knows we want to call the function.

The actual syntax of the print() function is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False
```

There's a few things we need to talk about here, as there's a lot of concepts that will apply not just to print, but many different functions and methods.

Authors note:
Don't forget about

```
print(help("print"))
```

print *(Built in function)* - `*objects`

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False
```

***objects argument,**
it's something that's quite useful in, the asterisk * denotes multiple values can be given at this point. The values themselves are collected into a Tuple. So the code can handle them appropriately.

Each argument is separated by a single comma "," then, in this case, their combined using the sep=' ' value, which is defaulted to a single whitespace.

```
print("James","Barrett", 123)
```

Technically, we are not joining or concatenating the strings. The function is constructing a new string on our behalf

print *(Built in function)* - Default parameters

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False
```

`sep=' ', end='\n',`
No doubt you've noticed the sep=' ', end='\n', These are example of default parameters. That is the say, if we do not specify them. The default values will be used. This is why sometimes you do not need to provide arguments.

To overwrite the default values (or Keyword arguments) we must use the parameter names,
To to overwrite the default values we must use the parameter names.
Note will have to put them at the end

```
print("James","Barrett",sep="-")
```

And the end dictates what the last characters printed is, as you'll see the default to a new line but if you wanted a change this behaviour, We can do so.

```
print("James",end=" ")
print("Barrett")
```

Authors note:
These can the Keyword arguments values strings of any length

print *(Built in function)*

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False
```

file=sys.stdout
The file is the object where the values are printed, and its default value is sys.stdout (screen).

```
fp= open("file.txt","w")
print("Hello File",file=fp)
fp.close()
```

flush : (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

Output Formatting String

Most of the time we need to format output instead of merely printing space-separated values. For example, we want to display the output left justified or in the centre or we want to show the number in the various formats.

You can display output in various styles and formats using the following methods.

- format()
- F String

format *(Multiple Placeholders - Positional Arguments)*

Multiple pairs of curly braces can be used while formatting the string.

Let's say if another variable substitution is needed in the sentence, it can be done by adding a second pair of curly braces and passing a second value into the method.

Python will replace the placeholders with values in the order in which their past into it.

```
name = "James"
age = 21
print("Hello {} I can't believe you {}".format(age, name))
```

You will need to have a value for every placeholder {} (substring) within the string object, otherwise it'll raise an an index error. (However, you do have to have a place holder for a value)

```
name = "James"
age = 21
description ="handsome"
print("Hello {} I can't believe you {}".format(name, age, description))
```

format *(Method in the string object)*

```
S.format(*args,**kwargs) ->str
```

To access the format method within the string, We use the dot operator at the end of the string object.

Remember Strings are immutable, as such, the format method will have to return a new string object when called.

Within the string, We use opening and closing curly braces as {*placeholders*} for values to be inserted, As such, any curly braces are placed with the provided values.

```
name = "James"
message= "Hello {}".format(name)
print(message)


name = "James"
print("Hello {}".format(name))


print("Hello {}".format("James"))
```

The value, is what we wish to put into the placeholders and concatenate with the string passed as parameters into the format function. The Value can be an Integer, Float, string, characters or even variables.

Authors note:
*args are essentially Tuples
and *kwargs, essentially dictionaries.

Don't forget about

```
print(help("str.format"))
```

```
str.format = format(...)
    S.format(*args, **kwargs) -> str

    Return a formatted version of S, using substitutions from args and kwargs.
    The substitutions are identified by braces ('{' and '}').
```

format *(Multiple Placeholders - Keyword Arguments & indexing)*

The format method also supports keyword arguments, as such, we can place the values in any order, but identify them by provided variable name.

The values could be literal (actual data or they could be references to other variables.

```
print("Hello {name} I can't believe you {age}".format(age = 21,name = "James"))

description ="Handsome"
print("Hello {name} I can't believe you {age}".format(age = 21,name = description))
```

As the arguments are pass in as Tuples, we can index the individual values, as well.

```
name = "James"
age = 21
print("Hello {1} I can't believe you {0}".format(age, name))
```

S.format (Format Specifier)

Each placeholder contains the Format specification to define how the value should be presented.

The general structure of standard format specifier is:

```
{field: [fill, align], sign #, width, group, .precision, type}
```

fill	This is the character added to pad string to the minimum length, ***(it has to be used with alignment)***
align	< (Align left), > (align right), ^ (align centre), = (align centre floats)
sign	Shows sign of integer value (+, -)
width	Minimum width , a must be positive integer
group	Can be either comma , or an underscore _ .will divide thousands when hundreds and so on
.precision	Signifies how many digits will be shown after a decimal value, or truncates the string
type	Specifies the data type of the value,

S.format (Format Specifier - **type**)

```
{field: [fill, align], sign #, width, group, .precision, type}
```

Type	
d	Single integer decimal
e	Floating point exponent
E	Floating point exponent
f	Float value
F	Float value
s	String
%	No conversion, results in percentage sign in the result

```
print("{:d}".format(123))
print("{:e}".format(123))
print("{:E}".format(123))
print("{:s}".format("james"))
```

S.format (Format Specifier - **.precision**)

```
{field: [fill, align], sign #, width, group, .precision, type}
```

The precision is the second last parameter and starts with a point .

It's important to note, it will actually round appropriately and not just be truncated, decimal values.

It can also be used to shorten (truncate) strings. This can be useful when there is a maximum number of characters that can be displayed.

```
print('{:.2f}'.format(122.403))
print('{:.2f}'.format(122.409))
print('{:.2f}'.format(122))
print('{:.0f}'.format(122))


print('{:.3}'.format("James"))


print('{:.<6.3}'.format("James"))
```

S.format (Format Specifier - **group**)

```
{field: [fill, align], sign #, width, group, .precision, type}
```

Minimum width , a must be positive integer.
The string will be padded with the fill character.

```
print('{:,d}'.format(1000000))
print('{:_d}'.format(1000000))

print('{:,.2f}'.format(1000000))
print('{:_.2f}'.format(1000000))
```

S.format (Format Specifier - **width**)

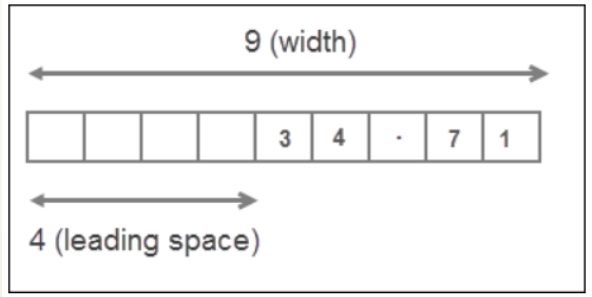
```
{field: [fill, align], sign, #, width, group, .precision, type}
```

This sets the Minimum width , it a must be positive integer.

The string will be padded with the film character.

Note if the value is longer than the minimum width, it will simply expand to include it.

```
print('€{:9.2f}'.format(34.71))
print('€{:2.2f}'.format(34.71))

print('€{:*>9.2f}'.format(34.71))
```



S.format (Format Specifier - #)

```
{field: [fill, align], sign, #, width, group, .precision, type}
```

This is used to specify any other number system than decimal.

```
print("bin {0:#b} ".format(5))
print("oct {0:#o} ".format(5))
print("hex {0:#x}".format(5))
```

S.format (Format Specifier - **sign**)

```
{field: [fill, align], sign, #, width, group, .precision, type}
```

Shows sign of integer value (+, -)

```
print('{:+d}'.format(-100))
print('{:+d}'.format(100))

print('{:-d}'.format(-100))
print('{:-d}'.format(100))
```

S.format (Format Specifier - **[fill, align]**)

```
{field: [fill, align], sign, #, width, group, .precision, type}
```

By default the fill value is a single whitespace

```
print('€{:*>9.2f}'.format(34.71))
print('{:.<6.3}'.format("James"))
print(':{:*^10}'.format("James"))
```

S.format (Format Specifier - **[fill, align]**)

```
{field: [fill, align], sign, #, width, group, .precision, type}
```

fill sets the minimum width that the value can be, the alignment is used to specify where the padding should be added.

```
print('€{:*>9.2f}'.format(34.71))
print('{:.<6.3}'.format("James"))
print(':{:*^10}'.format("James"))
```

F-strings

PEP 498 introduced a new string formatting mechanism known as Literal String Interpolation or more commonly as F-strings (because of the leading f character preceding the string literal).

The idea behind f-strings is to make string interpolation simpler.
To create an f-string, prefix the string with the letter " f ".

The string itself can be formatted in much the same way that you would with str.format().

F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.

```
name = "James"
print("Hello {}".format(name))
```

```
name = "James"
print(f"Hello {name}")
```

F-strings

Every f-string statement consists of two parts, one is character f or F, and the next one is a string which we want to format.

The string will be enclosed in single, double, or triple quotes.

In place of string, we must place our sentence which is going to formatted.

```
name = "James"
print(f"Hello {name}")

print(f'Hello {name}')

print(f'''Hello {name}
    how are you
    ''')


print(f"""Hello {name}
    how are you
    """)

print(rf"Hello \n {name}")
```

Python executes statements one by one and once f-string expressions are evaluated; they don’t change even if the expression value changes.

That’s why in the below code snippets, f_string value remains the same even after ‘name’ and ‘age’ variable has changed in the later part of the program.

```
name = "James"
message= f"Hello {name}"
print(message)

name = "James Barrett"
print(message)
```

f-string format floats

Floating point values have the f suffix. We can also specify the precision: the number of decimal places. The precision is a value that goes right after the dot character.

```
name="James"
cost=25.00

print(f'€{cost:*>9.2f}')
print(f'{name:.<6.3}')
print(f':{name:*^10}')
```