UNIVERSITY OF ENGINEERING AND TECHNOLOGY PESHAWAR, JALOZAI CAMPUS

Lab 3: Basic Python Data types, Built in Function and Function

Lab Title: EE-271 "OOP & Data Structures Lab"

Note: Using the internet is encouraging for finding relevant code and modifying it for the problem at hand.

Time: 10 min/ Task

Lab Report Tasks:

i. Define a function that will take the parameters as current and resistance and will output the voltage across the resistance.

   a. The argument pass will like positional arguments.

   b. The arguments passing will like keyword arguments.

   c. The arguments passing will like default arguments.

   d. Pass the arguments of one style to the other.

   e. Passing less number of arguments and check the error message.

   f. Verify that whether positional argument follows keyword argument.

Lab work tasks:

Sting formatting:

1. Observe the output of the following if:

```
errno = 50159747054
`name = 'Bob'
```

   a. `'Hello, %s' % name`

   b. `'Hey %s, there is a 0x%x error!' % (name, errno)`

   c. `'Hey %(name)s, there is a 0x%(errno)x error!' % { "name": name, "errno": errno }`

2. Observe the output of: (Mostly used)

   a. `'Hello, {}'.format(name)`

3. Observe the output of: (Recommended for Python 3.6+)

   a. `f'Hello, {name}!'`

4. Observe the output of:

   a.

```
from string import Template
```

```
t = Template('Hey, $name!')

st.substitute(name=name)
```

b.

```
templ_string = 'Hey $name, there is a $error error!'

Template(templ_string).substitute(name=name, error=hex(errno))
```

Built-in Function:

i. Use len() and find the length of three lists and two strings.

ii. Use for three example .is_integer().

iii. Use for three example .conjugate()

Function:

i. Define a function that will take the name and registration number of a student and print it. Include appropriate docstrings.

   a. The argument pass will like positional arguments.

   b. The arguments passing will like keyword arguments.

   c. The arguments passing will like default arguments.

   d. Pass the arguments of one style to the other.

   e. Passing less number of arguments and check the error message.

   f. Verify that whether positional argument follows keyword argument

ii. Define a function that will take the parameters as current and resistance and will output the voltage across the resistance. Include appropriate docstrings.

   a. The argument pass will like positional arguments.

   b. The arguments passing will like keyword arguments.

   c. The arguments passing will like default arguments.

   d. Pass the arguments of one style to the other.

   e. Passing less number of arguments and check the error message.

   f. Verify that whether positional argument follows keyword argument.

iii. Define a function with name charge_from_solar and pass the state of charge (SOC) to the function. If SOC is less than 20% display charging if it is above 90% do nothing and return and if it is in between display that battery will be charge and there is --- % of charge in the battery. Include

appropriate docstrings. Print the doc string. Also add annotation to the function parameters and return type.

iv. Define three function. Function one name is resistance which will return a list of five resistances. The second function name is voltage which will return a list of voltage across each of these resisters. Include appropriate docstrings. Print the doc string. Also add annotation to the function parameters and return type.

   a. Pass these two lists to a third function name current which will calculate the current flowing through each resister and will return that list to the main program. The main program will print all the three lists in well formatting.

   b. In part a return the data in tuple.

   c. Return the data from current function as dictionary with keys current_1 to current_5.

v. Define a function with name series. The function will take a list or tuple of resisters. The function will return the net resistance. Note that your function must be capable for any number of resisters. Note: Use elegant approach. Include appropriate docstrings. Print the doc string. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

vi. Define a function with name parallel. The function will take a list or tuple of resisters. The function will return the net resistance. Note that your function must be capable for any number of resisters. Note: Use elegant approach. Include appropriate docstrings. Print the doc string. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

vii. Define a function with name series. The function will take resisters values and use *args for tuple packing. The function will return the net resistance. Note that your function must be capable for any number of resisters. Note: Use elegant approach. Include appropriate docstrings. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

viii. Define a function with name parallel. The function will take resisters values and use *args for tuple packing. The function will return the net resistance. Note that your function must be capable for any number of resisters. Note: Use elegant approach. Include appropriate docstrings. Print the doc string. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

ix. Consider a scenario in which you receive the data of battery SOC (% age), solar generation (Kw) and market price (per unit in rupees) from a hybrid solar inverter that are connected with green meter in a list or tuple like [60, 2, 60] or (60, 2, 60). Design a function that will print these values in a well format. Note use the tuple or list unpacking. Include appropriate docstrings. Print the doc string. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

x. The more elegant approach is to pass dictionary and use dictionary unpacking. Consider a scenario in which you receive the data of battery SOC (% age), solar generation (Kw) and market price (per unit in rupees) from a hybrid solar inverter that are connected with green meter in a dictionary like {'SOC': 60, 'solar generation': 2, 'price': 60}. Design a function that will receive this dictionary and print these values in a well format. Include appropriate docstrings. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

xi. Change the scenario while keep the more elegant approach of pass dictionary and use dictionary packing. Consider a scenario in which you receive the data of battery SOC (% age), solar generation (Kw) and market price (per unit in rupees) from a hybrid solar inverter that are connected with green meter in a dictionary like {'SOC': 60, 'solar generation': 2, 'price': 60}. Let you pass these value in a function call like solar (SOC = 60, solar generation = 2, price = 60). Design a function that will receive these argument as a dictionary and print these values in a well format. Include appropriate docstrings. Also add annotation to the function parameters and return type. In addition, display annotation using special dunder attribute of the function called `__annotations__`.

xii. Understand the functions given below by run, modify and also design different scenarios.

```
def f(a, b, *args, **kwargs):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'args = {args}')
    print(F'kwargs = {kwargs}')


f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)
```

```
def f(*args):
    for i in args:
            print(i)


a = [1, 2, 3]
t = (4, 5, 6)
s = {7, 8, 9}
```

```
f(*a, *t, *s)
```

```
def f(**kwargs):
    for k, v in kwargs.items():
            print(k, '->', v)


d1 = {'a': 1, 'b': 2}
d2 = {'x': 3, 'y': 4}

f(**d1, **d2)
```

```
def f(*args):
    for i in args:
            print(i)


f(*[1, 2, 3], *[4, 5, 6])
```

```
def f(**kwargs):
    for k, v in kwargs.items():
            print(k, '->', v)


f(**{'a': 1, 'b': 2}, **{'x': 3, 'y': 4})
```

```
def f(a: int, b: str) -> float:
    print(a, b)
    return(3.5)
```

```
f(1, 'foo')
f.__annotations__
```

If you want to assign a default value to a parameter that has an annotation, then the default value goes after the annotation:

```
def f(a: int = 12, b: str = 'baz') -> float:
    print(a, b)
    return(3.5)

f.__annotations__

f()
```

Annotations don't impose any **semantic restrictions** on the code whatsoever.
What's going on here? The annotations for `f()` indicate that the first argument is `int`, the second argument `str`, and the return value `float`. But the subsequent call to `f()` breaks all the rules! The arguments are `str` and `float`, respectively, and the return value is a tuple. Yet the interpreter lets it all slide with no complaint at all.

```
def f(a: int, b: str) -> float:
    print(a, b)
    return 1, 2, 3

f('foo', 2.5)
```

String:

i. Use of `\n` and `\t`. It is encouraging to learn the use of the following table of escape sequences.

Escape Sequence	Escaped Interpretation
`\a`	ASCII Bell (BEL) character
`\b`	ASCII Backspace (BS) character
`\f`	ASCII Formfeed (FF) character
`\n`	ASCII Linefeed (LF) character
`\N{<name>}`	Character from Unicode database with given `<name>`
`\r`	ASCII Carriage return (CR) character
`\t`	ASCII Horizontal tab (TAB) character
`\uxxxx`	Unicode character with 16-bit hex value xxxx
`\Uxxxxxxxx`	Unicode character with 32-bit hex value xxxxxxxx
`\v`	ASCII Vertical tab (VT) character
`\ooo`	Character with octal value ooo
`\xhh`	Character with hex value hh

ii. Use F-String for three string formatting.

iii. Use `.capitalize()` for three examples.

iv. Use `.format(*args, **kwargs)` for five example formatting.

v. Use `.lower()` for three examples.

It is recommended to understand and use the following list of method in some examples. s

Method	Description
`.capitalize()`	Converts the first character to uppercase and the rest to lowercase
`.casefold()`	Converts the string into lowercase
`.center(width[, fillchar])`	Centers the string between `width` using `fillchar`
`.encode(encoding, errors)`	Encodes the string using the specified `encoding`
`.expandtabs(tabsize)`	Replaces tab characters with spaces according to `tabsize`
`.format(*args, **kwargs)`	Interpolates and formats the specified values
`.format_map(mapping)`	Interpolates and formats the specified values using a dictionary
`.join(iterable)`	Joins the items in an iterable with the string as a separator
`.ljust(width[, fillchar])`	Returns a left-justified version of the string
`.rjust(width[, fillchar])`	Returns a right-justified version of the string

`.lower()`	Converts the string into lowercase
`.strip([chars])`	Trims the string by removing `chars` from the beginning and end
`.lstrip([chars])`	Trims the string by removing `chars` from the beginning
`.rstrip([chars])`	Trims the string by removing `chars` from the end
`.removeprefix(prefix, /)`	Removes `prefix` from the beginning of the string
`.removesuffix(suffix, /)`	Removes `suffix` from the end of the string
`.replace(old, new [, count])`	Returns a string where the `old` substring is replaced with `new`
`.swapcase()`	Converts lowercase letters to uppercase letters and vice versa
`.title()`	Converts the first character of each word to uppercase
`.upper()`	Converts a string into uppercase
`.zfill(width)`	Fills the string with a specified number of zeroes at the beginning

vi. Some method on string return true false and they are frequently use in different applications. Use `.isdigit()`,`.islower()`, and `.isnumeric()` in two example. It is recommended to master all of these methods which are enlisted in the following table.

Method	Result
`.endswith(suffix[, start[, end]])`	`True` if the string ends with the specified suffix, `False` otherwise
`.startswith(prefix[, start[, end]])`	`True` if the string starts with the specified prefix, `False` otherwise
`.isalnum()`	`True` if all characters in the string are alphanumeric, `False` otherwise
`.isalpha()`	`True` if all characters in the string are letters, `False` otherwise
`.isascii()`	`True` if the string is empty or all characters in the string are ASCII, `False` otherwise
`.isdecimal()`	`True` if all characters in the string are decimals, `False` otherwise

`.isdigit()`	`True` if all characters in the string are digits, `False` otherwise
`.isidentifier()`	`True` if the string is a valid Python name, `False` otherwise
`.islower()`	`True` if all characters in the string are lowercase, `False` otherwise
`.isnumeric()`	`True` if all characters in the string are numeric, `False` otherwise
`.isprintable()`	`True` if all characters in the string are printable, `False` otherwise
`.isspace()`	`True` if all characters in the string are whitespaces, `False` otherwise
`.istitle()`	`True` if the string follows title case, `False`

vii. Some other string operation required other methods. Use `.split() for two example.`

It is recommended to master all of these methods which are enlisted in the following table.

Method	Description
`.count(sub[, start[, end]])`	Returns the number of occurrences of a substring
`.find(sub[, start[, end]])`	Searches the string for a specified value and returns the position of where it was found
`.rfind(sub[, start[, end]])`	Searches the string for a specified value and returns the last position of where it was found
`.index(sub[, start[, end]])`	Searches the string for a specified value and returns the position of where it was found
`.rindex(sub[, start[, end]])`	Searches the string for a specified value and returns the last position of where it was found
`.split(sep=None, maxsplit=-1)`	Splits the string at the specified separator and returns a list
`.splitlines([keepends])`	Splits the string at line breaks and returns a list
`.partition(sep)`	Splits the string at the first occurance of `sep`
`.rpartition(sep)`	Splits the string at the last occurance of `sep`
`.split(sep=None, maxsplit=-1)`	Splits the string at the specified separator and returns a list
`.maketrans(x[, y[, z]])`	Returns a translation table to be used in translations
`.translate(table)`	Returns a translated string

Operation	Example	Result
Length	`len(s)`	The length of `s`
Indexing	`s[index]`	The item at index `i`
Slicing	`s[i:j]`	A slice of `s` from index `i` to `j`
Slicing	`s[i:j:k]`	A slice of `s` from index `i` to `j` with step `k`
Minimum	`min(s)`	The smallest item of `s`
Maximum	`max(s)`	The largest item of `s`
Membership	`x in s`	`True` if an item of `s` is equal to `x`, else `False`
Membership	`x not in s`	`False` if an item of `s` is equal to `x`, else `True`
Concatenation	`s + t`	The concatenation of `s` and `t`
Repetition	`s * n` or `n * s`	The repetition of `s` a number of times specified by `n`
Index	`s.index(x[, i[, j]])`	The index of the first occurrence of `x` in `s`
Count	`s.count(x)`	The total number of occurrences of `x` in `s`