UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

# Topic 4:

# Strings (Part 2)

CSGE601020 - Dasar-Dasar Pemrograman 1

Lintang Matahari Hasani, S.Kom., M.Kom. | Dr.Eng. Lia Sadita, S.Kom., M.Eng.

# Acknowledgement

This slide is an adapted version of ‘Strings’ slides used in DDP1 Course (2020/2021)
by **Hafizh Rafizal Adnan, M.Kom.**

Several materials are reused from ‘Strings’ slides used in Dasar-Dasar Pemrograman 1 dengan Python (CSGE601020/4 SKS) Course (https://ocw.ui.ac.id/course/view.php?id=142)
by **Fariz Darari, Ph.D.**



Some additional contents, illustrations and visual design elements are provided by **Lintang Matahari Hasani, M.Kom.**

# In this session, you will learn ...

Part 1

- String Type
- Unicode
- String Slicing and Operations
- A Preview of String Function and Methods

Part 2

- Formatted Output for Strings
- String Modulo Operator
- f-String

# String Formatting (1)

- ➔ So far, we have just used the defaults of the print function
- ➔ We can do many more complicated things to make that output more **customized**.
- ➔ We will use it in our display function

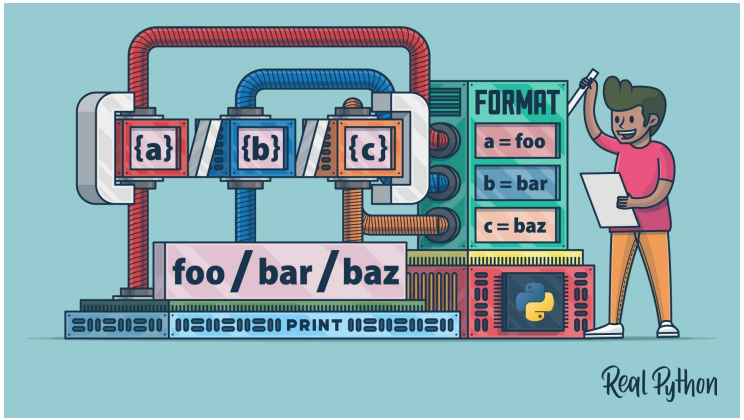



Image credits to: https://realpython.com/python-formatted-output/

# String Formatting (2): Motivation

```
masukkan nilai bunga deposito (%):2.5
masukkan saldo awal deposito:100000000
masukkan jangka waktu deposito (bulan):15
Bulan ke-n | Saldo Akhir Bulan ke-n
-------------------------
|Bulan ke- 0   |  Saldo = Rp. 102500000.0   |
|Bulan ke- 1   |  Saldo = Rp. 105062500.0   |
|Bulan ke- 2   |  Saldo = Rp. 107689062.5   |
|Bulan ke- 3   |  Saldo = Rp. 110381289.0625   |
|Bulan ke- 4   |  Saldo = Rp. 113140821.2890625   |
|Bulan ke- 5   |  Saldo = Rp. 115969341.82128906   |
|Bulan ke- 6   |  Saldo = Rp. 118868575.36682129   |
|Bulan ke- 7   |  Saldo = Rp. 121840289.75099182   |
|Bulan ke- 8   |  Saldo = Rp. 124886296.99476662   |
|Bulan ke- 9   |  Saldo = Rp. 128008454.41963579   |
|Bulan ke- 10  |  Saldo = Rp. 131208665.78012668   |
|Bulan ke- 11  |  Saldo = Rp. 134488882.42462984   |
|Bulan ke- 12  |  Saldo = Rp. 137851104.4852456   |
|Bulan ke- 13  |  Saldo = Rp. 141297382.09737673   |
|Bulan ke- 14  |  Saldo = Rp. 144829816.64981115   |
-------------------------
```

```
masukkan nilai bunga deposito (%):2.5
masukkan saldo awal deposito:100000000
masukkan jangka waktu deposito (bulan):15
| Bulan ke-n |          Saldo Akhir Bulan ke-n          |
-------------------------------------------------------
|Bulan ke-0  |          Saldo = Rp.102,500,000.00       |
|Bulan ke-1  |          Saldo = Rp.105,062,500.00       |
|Bulan ke-2  |          Saldo = Rp.107,689,062.50       |
|Bulan ke-3  |          Saldo = Rp.110,381,289.06       |
|Bulan ke-4  |          Saldo = Rp.113,140,821.29       |
|Bulan ke-5  |          Saldo = Rp.115,969,341.82       |
|Bulan ke-6  |          Saldo = Rp.118,868,575.37       |
|Bulan ke-7  |          Saldo = Rp.121,840,289.75       |
|Bulan ke-8  |          Saldo = Rp.124,886,296.99       |
|Bulan ke-9  |          Saldo = Rp.128,008,454.42       |
|Bulan ke-10 |          Saldo = Rp.131,208,665.78       |
|Bulan ke-11 |          Saldo = Rp.134,488,882.42       |
|Bulan ke-12 |          Saldo = Rp.137,851,104.49       |
|Bulan ke-13 |          Saldo = Rp.141,297,382.10       |
|Bulan ke-14 |          Saldo = Rp.144,829,816.65       |
-------------------------------------------------------
```

➔ Without formatting

➔ In this session, we will learn how to format output like this using `format()` and string modulo operator (C-style formatting)

# String Formatting: Basic Form

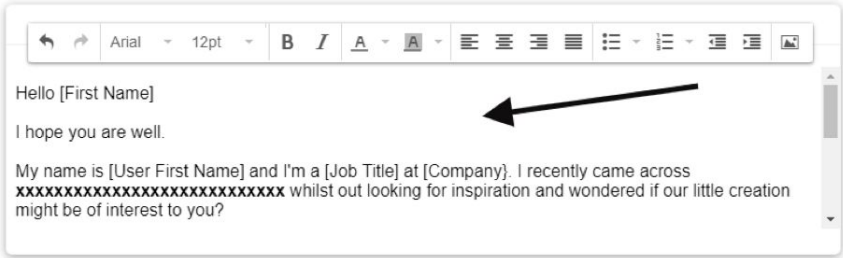
To understand string formatting, it is probably better to start with an example.
Suppose we want to print a string with a certain template: **bla bla bla *{your_values_here}* bla bla bla**

```
print("Sorry, is this the {} minute {}?".format(5, 'ARGUMENT'))
```

Output:

```
Sorry, is this the 5 minute ARGUMENT?
```

It's like writing an email with
a predetermined template ^^



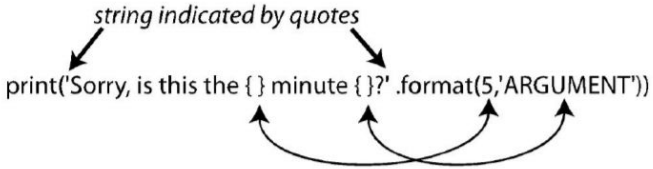
# String Formatting: `format()` Method

**`format`** is a method that creates a new string where certain elements of the string are **re-organized**

General form: `<template>.format(<positional_argument(s)> or <keyword_argument(s)>)`

The `<template>` string includes the **replacement fields** indicated by **curly brackets `{ }`**

The elements to be re-organized are the curly bracket elements in the string.




- ➔ The string is modified so that **the `{}` elements** in the string are **replaced by the `format()` method positional/keywords arguments.**

- ➔ **Default since Python 3.1**: **Automatic field numbering** The replacement is **in order**: first `{}` is replaced by the first argument, second `{}` by the second argument and so forth.

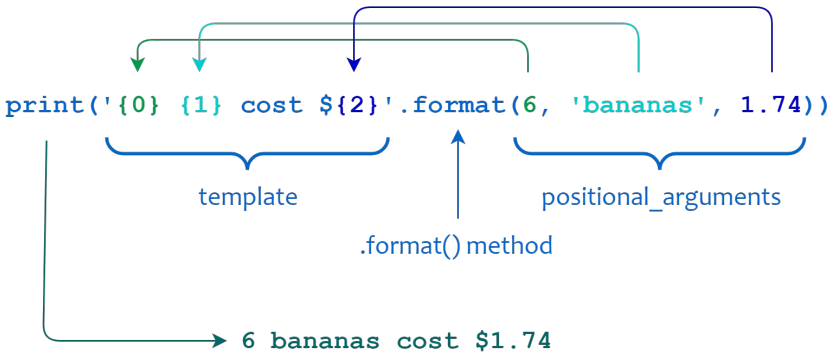
# `format()` Method Using `<positional_arguments>`

**Using positional arguments:**

```
<template>.format(<positional_argument(s)>)
```

```
print('{0} {1} cost ${2}'.format(6, 'bananas', 1.74))
```

```
>>> print('{0} {1} cost ${2}'.format(6, 'bananas', 1.74))
6 bananas cost $1.74
```




https://realpython.com/python-formatted-output/#the-python-string-format-method

# `format()` Method Using `<positional_arguments>`

By using positional arguments, we don't need to arrange the replacement fields in numerical order. We can specify them **in any order**.

example:

```
print('{3}: {0} {1} {2}'.format('Veritas', 'Probitas', 'Iustitia', 'Motto'))
```

```
>>> print('{3}: {0} {1} {2}'.format('Veritas', 'Probitas', 'Iustitia', 'Motto'))
Motto: Veritas Probitas Iustitia
```

**Warning**: do not specify a replacement field with a number that is out of range.
This will produce error (`IndexError`) ^^

# What does this program do?

Write the output in the **comment section**

```
item = 'Genesis Crystals'
jumlah = 300
harga = 75000
currency = 'Rp.'

print('{2} {1}: {3}{0}'.format(harga, item ,jumlah ,currency))
```

# `format()` Method with Automatic Field Numbering

Since Python 3.1, we can **omit the ordering numbers** in the replacement fields.
By default, the replacement is in order: first `{}` is replaced by the first argument, second `{}` by the second argument and so forth.

example:

```
print('{}: {} {} {}'.format('Motto', 'Veritas', 'Probitas', 'Iustitia'))
```

```
>>> print('{}: {} {} {}'.format('Motto', 'Veritas', 'Probitas', 'Iustitia'))
Motto: Veritas Probitas Iustitia
```

**Warning**: The specified replacement fields must be <= the number of arguments provided. Otherwise, this will produce error (`IndexError`).

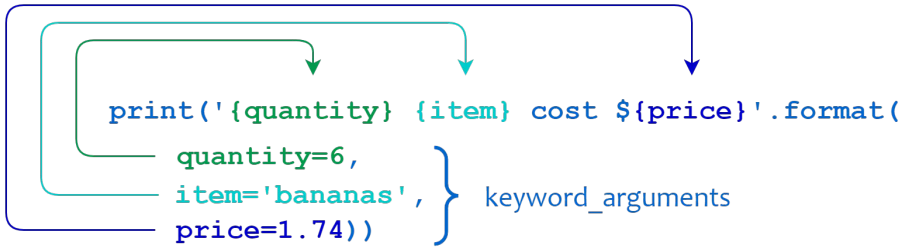
However, if the number of replacement fields < the number of arguments provided, no error will occur ^^

# `format()` Method Using `<keyword_arguments>`

**Using keyword arguments:**

```
<template>.format(<keyword_argument(s)>)
```

```
print('{quant} {item} cost ${price}' .format(quant = 6, item = 'bananas', price =
1.74
>>> print('{quant} {item} cost ${price}'.format(quant = 6, item = 'bananas', price = 1.74))
6 bananas cost $1.74
```




**Warning**: do not specify a replacement field with a missing keyword. This will produce an error (`KeyError`) ^^

https://realpython.com/python-formatted-output/#the-python-string-format-method

**Triggering Question 2**

# What does this program do?

Write the output in the **comment section**

```
item = 'Genesis Crystals'
jumlah = 1980
harga = 439000
currency = 'Rp.'

print('{} {}: {}{}. Get {} now!' .format(jumlah, item ,currency ,harga,  '1980 primogems'))
```

1

```
item = 'Genesis Crystals'
harga = 75000
currency = 'Rp.'

print('{} {}: {}{}. Get {} now! {}' .format(300, item ,currency ,harga,  '300 primogems'))
```

2

# String Formatting: The Anatomy of a Replacement Field

Each bracket `{replacement_field}` looks like:

```
{<name> <conversion> : <format_specification>]}
```

```
print('Jumlah mhs: {jml_mhs!s:^10s} orang' .format(jml_mhs = 2000))
```

➔ `<name>` indicates the **argument** to be inserted (can be omitted, is an ordering number/a keyword)

```
print('Jumlah mhs: {jml_mhs!s:^10s} orang'.format(jml_mhs = 2000))
```

➔ `<conversion>` enables conversion **to string**

```
print('Jumlah mhs: {jml_mhs!s:^10s} orang'.format(jml_mhs = 2000))
```

➔ `<format_specification>` is for formatting the values (i.e., **width**, **rounding**, **alignment**)

```
print('Jumlah mhs: {jml_mhs!s:^10s} orang'.format(jml_mhs = 2000))
```

# Format Specification: Basic Form

The **<format_specification>** inside a **{replacement_field}** looks like:

```
{<name> <conversion> :align width .precision descriptor}
```

- ➔ `align` is optional (default left for strings, right for numbers)
- ➔ `width` is how many spaces (default just enough)
- ➔ `.precision` is for floating point rounding (default norounding)
- ➔ `descriptor` is the expected type (error if the arg is the wrong type)

an example:

```
print('Suhu tubuh: {suhu :^10.2f} °C'.format(suhu = 37.5))
```

```
>>> print('Suhu tubuh: {suhu:^10.2f} °C'.format(suhu = 37.5))
Suhu tubuh:   37.50    °C
```

align: ^ (**center**)
width: 10 **spaces** wide including the object

precision: .2 (**2 number of digits** after the decimal point)
descriptor: f (the expected type is a **float**)

# String Formatting: Format String (2)

The content of the curly bracket elements are the format string, descriptors of how to organize that particular substitution.

➔ Types are the kind of thing to substitute, numbers indicate total spaces

| | |
|---|---|
| s | string |
| d | decimal integer |
| f | floating-point decimal |
| e | floating-point exponential |
| % | floating-point as percent |

**TABLE 4.3** Most commonly used types

| | |
|---|---|
| < | left |
| > | right |
| ^ | center |

**TABLE 4.4** Width alignments.

```
>>> print('Suhu tubuh: {suhu:<10.2f} °C'.format(suhu = 37.5))
Suhu tubuh: 37.50      °C
>>> print('Suhu tubuh: {suhu:>10.2f} °C'.format(suhu = 37.5))
Suhu tubuh:      37.50 °C
>>> print('Suhu tubuh: {suhu:^10.4f} °C'.format(suhu = 37.5))
Suhu tubuh:  37.5000   °C
```

# String Formatting: Format String (3)

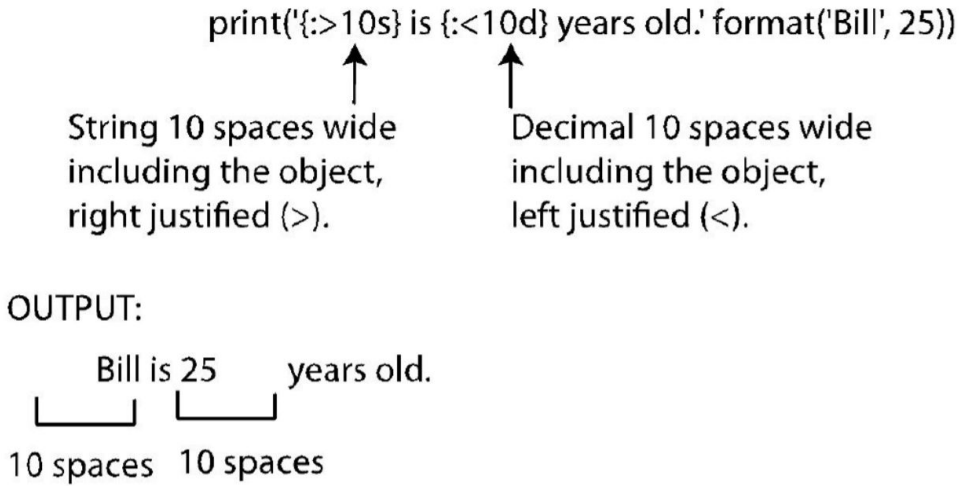



**FIGURE 4.11** String formatting with width descriptors and alignment.

# What does this program do?

Write the output in the **comment section**

```
print('Suhu: {suhu:<.4f} °C'.format(suhu = 37.5789912))
```

1

```
print('Suhu: {suhu:^.4d} °C'.format(suhu = 39.12))
```

2

```
print('Suhu: {suhu:>10.4f} °C'.format(suhu = 40.125673))
```

3

# Advanced Formatting (1)

We have discussed that a format string was of the following form:

```
{<name> <conversion> :align width .precision descriptor}
```

Welp, it can be more complicated than that. The full form of the <format_specification> is specified below.

```
{<name> <conversion> : fill align sign # 0 width , .precision descriptor}
```

That's a lot, so let's look at the details

# Advanced Formatting (2)

```
{<name> <conversion> : fill align sign # 0 width , .precision descriptor}
```

**fill**

Besides alignment, you can fill empty spaces with a fill **character**:

- 0 = fill with 0's
- \+ = fill with +
- . = fill with .
- $ = fill with $

```
print('{0:$>40s}'.format('di sebelah kiri ada 10 tanda $'))
```

```
print('{0:0>10d}'.format(234))
```

**sign**

- \+ means a sign for both positive and negative numbers
- – means a sign for only negative numbers
- *space* means space for positive, minus for negative

```
print('{0:-.2f}'.format(29.259))
```

```
29.26
```

# Advanced Format Method (3)

```
{<name> <conversion> : fill align sign # 0 width , .precision descriptor}
```

- ➔ **#** Forces **an alternate output form** for certain presentation types (e.g., #6.0, a decimal point)
- ➔ **0** Causes values to be padded **on the left with zeros** instead of ASCII space characters
- ➔ **,** put **commas every three digits**

```
>>> print('{:#6.0f}'.format(3)) # decimal point forced
    3.
>>> print('{:04d}'.format(4))  # zero preceeds width
0004
>>> print('{:,d}'.format(1234567890))
1,234,567,890
```

# Advanced Format Method: Example

```
print('{0:.>12s} | {1:0=+10d} | {2:->5d}' .format('abc', 35, 22))
```

for example `{1:0=10d}` means:

- `1` ☐ positional argument: **get the second object** (count from 0), that is the integer 35
- `:` ☐ separator that separates `<name> <conversion>` with the `<format_specification>`
- `0=` ☐ fill with 0's
- `+` ☐ plus or minus sign
- `10` ☐ occupy 10 spaces (default: left justify, there is no alignment symbol ^, <, or > stated in the expression)
- `d` ☐ the expected object type to be printed is decimal

```
>>> print('{0:.>12s} | {1:0=+10d} | {2:->5d}'.format('abc', 35, 22))
.........abc | +000000035 | ---22
```

# What does this program do?

Write the output in the **comment section**

```
print('{0:*<10s} | {1:0=-10d} | {2:@>4d}' .format('xyz', 777, 22))
```

Is the second integer printed as negative? Why?

# C-Style Formatting (1): Using Modulo Operator

It uses **%** sign rather than `{}`
It's easier and more commonly used

```
feet = 5
inches = 2
meters = 1.54779

print ("%d feet and %d inches is %1.4f meters" % (feet, inches, meters))
```

if feet = 5 and inches = 2, prints:

**`5 feet and 2 inches is 1.5748 meters`**

same specifiers - **%d** for integer (decimals), **%f** for float and **%s** for string

# C-Style Formatting (2)

**Width modifiers** come before field specifiers

```
feet = 5
inches = 2
print("%6d feet and %6d inches" % (feet, inches))
```

It prints the following characters (Note: **- is a space**):

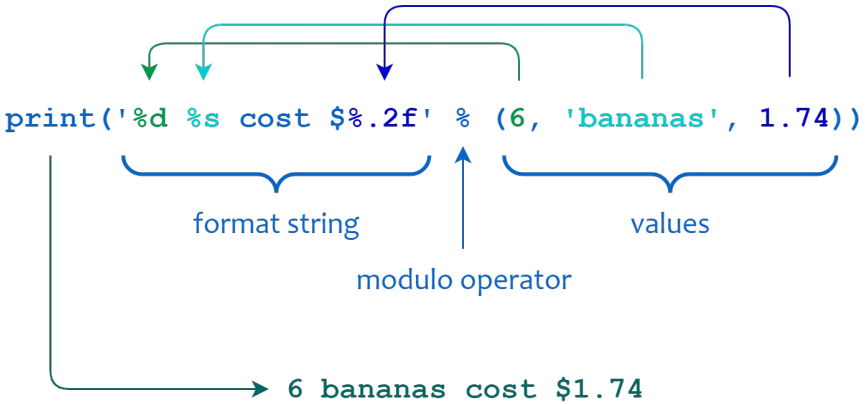
```
-----5 feet and -----2 inches
```

standard is right justified. If you want left, make the width modifier negative

# C-Style Formatting (3)

```
print('%d %s cost $%.2f' % (6, 'bananas', 1.74))
```

```
6 bananas cost $1.74
```




The '%' character also denotes the conversion specifiers in the format string—in this case, '%d', '%s', and '%.2f'

- The first item in the tuple: **6**, a numeric value that replaces **%d** in the format string.
- The next item: **'bananas'**, which replaces **%s**.
- The last item: **1.74**, which replaces **%.2f**

https://realpython.com/python-input-output/#the-string-modulo-operator

# C-Style Formatting (4): The Anatomy of C-Style Formatting

The general form looks like:

```
<format_string> % <values>
```

The <format_string> form looks like:

```
%<flags> <width> .<precision> <type>
```

```
print('6 bananas cost $%010.2f right now'  % (1.74))
```

➔ **<flags>** allow finer control over the **display** of certain conversion types (i.e., add 0's, justify left alignment, etc.)

```
print('6 bananas cost $% 010.2f right now.' % (1.74))
```

➔ **<width>** specifies the **minimum width** of the output field (default alignment: right justified)

```
print('6 bananas cost $%0 10.2f right now.' % (1.74))
```

➔ **<precision>** is for formatting the values with certain **<type>** (i.e., **number of digits after decimal point**)

```
print('6 bananas cost $%010 .2f right now.' % (1.74))
```

**More info:** *https://realpython.com/python-input-output/#the-string-modulo-operator*

# Formatting Table (Format Method)

```
>>> for n in range(3,11):
    print('{:4}-sides:{:6}{:10.2f}{:10.2f}'.format(n,180*(n-2),180*(n-2)/n,360/n)
```

```
  3-sides:   180     60.00    120.00
  4-sides:   360     90.00     90.00
  5-sides:   540    108.00     72.00
  6-sides:   720    120.00     60.00
  7-sides:   900    128.57     51.43
  8-sides:  1080    135.00     45.00
  9-sides:  1260    140.00     40.00
 10-sides:  1440    144.00     36.00
```

# Formatting Table (C-Style Formatting)

```
for n in range(3,11):
    print ("%4d-sides:%6d%10.2f%10.2f" %
  (n, 180*(n-2),180*(n-2)/n, 360/n) )
```

```
 3-sides:  180    60.00   120.00
 4-sides:  360    90.00    90.00
 5-sides:  540   108.00    72.00
...
```

# Back to Basic: `sep` Argument

- ➔ Function `print()` takes 0 or more arguments and prints them in the shell
- ➔ A blank space **separator** is **printed between the arguments**
- ➔ The `sep` argument allows for customized separators

```
pets = ['boa', 'cat', 'dog']

for pet in pets:
    print(pet)

for pet in pets:
    print(pet, end=', ')

for pet in pets:
    print(pet, end='!!! ')
```

# A Faster Way for String Formatting: f-String

- ➔ Using the “old school” techniques (`str.format()` and `%`) for placing certain values in a specific place inside a string are usually quite complex
- ➔ Since Python 3.6, we can use f-string
- ➔ You can type specifier in the f-string just like when you use `str.format()`

```
print("Sorry, is this the {} minute {}?".format(5, 'ARGUMENT'))
```

Using `format()`

```
print("Sorry, is this the %1d minute %8s?"%(5, 'ARGUMENT'))
```

Using C-style formatting

```
minute = 5
argument = 'ARGUMENT'
print(f"Sorry, is this the {minute} minute {argument}?")
```

Using f-string

```
jml_mhs = 2000
print(f'Jumlah mhs: {jml_mhs!s:^10s} orang')
```

Using f-string
with a type specifier

# Review Questions (Part 1)

1. What are the outputs?

```
x = 'leo'
y = 'nico'
z = 'leon'

print(z in (x + y) * 3)
print(z in (x[::-1] + y[1 : 3]))
```

2. What is the output?

```
chr('DDPKuSuka')
```

# Review Questions (Part 2)

Suppose `s` is assigned as follows:

```python
s = 'foobar'
```

All of the following expressions produce the same result except one. Which one?

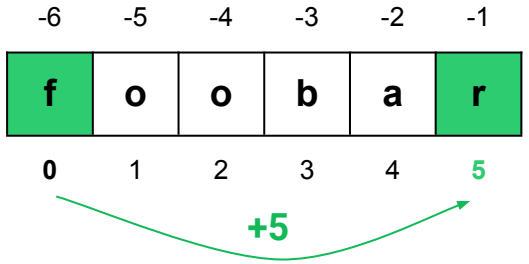
- `s[::5]`
- `s[::-1][-1] + s[len(s)-1]`
- `s[0] + s[-1]`
- `s[::-1][::-5]`
- `s[::-5]`

https://realpython.com/quizzes/python-strings/

# Explanation for Review Question Part 2

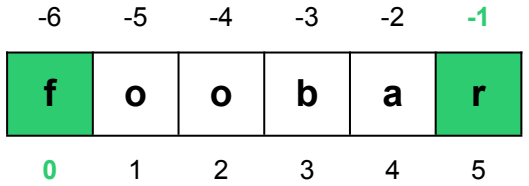
```
s = 'foobar'
```

```
s[::5]
```




`[::5]` specifies every fifth character, starting at the beginning and proceeding to the end (because the first two indices are allowed to default). Thus, the result is the first character, `'f'`, followed by the sixth character, `'r'`.

```
s[0] + s[-1]
```

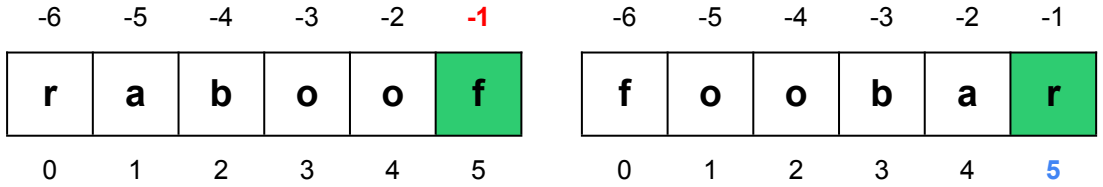



`s[0]` is the first character, `'f'`.
`s[-1]` is the last character, `'r'`.

# Explanation for Review Question Part 2

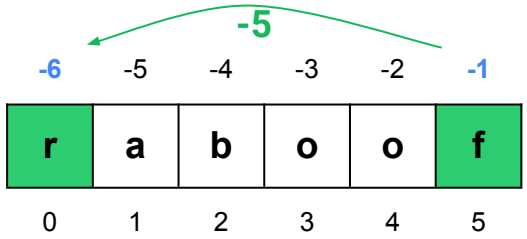
```
s = 'foobar'
```

```
s[::-1][-1] + s[len(s)-1]
```




`s[::-1]` **reverses** the string, so it equals `'raboof'`.
The added `[-1]` index specifies the last character of that string, `'f'`.
`s[len(s)-1]` is the same as `s[-1]` — the last character of the original string, `'r'`.

```
s[::-1][::-5]
```




As above, `s[::-1]` is `'raboof'`. `[::-5]` specifies **every fifth character**, **starting at the end** and **proceeding to the beginning** (remember that when the stride is negative, the first index defaults to the end of the string, and the second index to the beginning, rather than the other way around). Thus, this returns `'f'`, then `'r'`.

# Review Questions (Part 3)

Fix (Rapikan ^^) the output of this program:

```
saldo_akhir_bulan = 100000000
n = 12
bunga = 2.5
for bulan in range(0,n):
    saldo_akhir_bulan = saldo_akhir_bulan + (saldo_akhir_bulan * (bunga/100))
    print('|Bulan ke-', bulan, ' | ' , 'Saldo = Rp.', saldo_akhir_bulan, '  |')
```

So that it will produce the following output:

```
|Bulan ke- 0  |Saldo = Rp.102,500,000.00      |
|Bulan ke- 1  |Saldo = Rp.105,062,500.00      |
|Bulan ke- 2  |Saldo = Rp.107,689,062.50      |
|Bulan ke- 3  |Saldo = Rp.110,381,289.06      |
|Bulan ke- 4  |Saldo = Rp.113,140,821.29      |
|Bulan ke- 5  |Saldo = Rp.115,969,341.82      |
|Bulan ke- 6  |Saldo = Rp.118,868,575.37      |
|Bulan ke- 7  |Saldo = Rp.121,840,289.75      |
|Bulan ke- 8  |Saldo = Rp.124,886,296.99      |
|Bulan ke- 9  |Saldo = Rp.128,008,454.42      |
|Bulan ke- 10 |Saldo = Rp.131,208,665.78      |
|Bulan ke- 11 |Saldo = Rp.134,488,882.42      |
```

- ➔ The “Bulan ke-n” string occupies **10 spaces with** left **alignment**
- ➔ The “Saldo = Rp. [values]” must be formatted with **commas every 3 zeros**, occupies **20 spaces** with **left alignment**, and **round the float until 2 digits** after the decimal point
- ➔ You may use `format()` or C-style formatting

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

# Q&A Session