UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Topic 4:
Strings

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

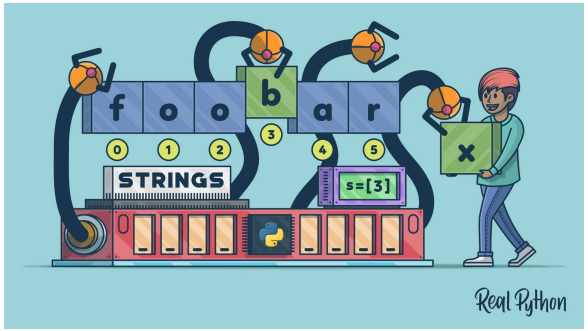
- Formatted Output for Strings
- String Modulo Operator

String: A Sequence of characters

We've talked about strings being **a sequence of characters**.

A string is indicated between ' ' *or* " "

- ➔ The exact sequence of characters is maintained
- ➔ It is also a collection



Create the object with assignment:

```
sample_1 = "Todoroki Shouto"
```

Create the object or `str` constructor:

```
sample_2 = str("Todoroki Shouto")
```

Constructing a String

Using **single** or **double quotes**:

```
sample1 = 'Todoroki Shouto'

sample2 = "Todoroki Shouto"
```

Do not mix them as shown below;

```
sample3 = 'Gojo Satoru" #ERROR!
```

Inserting an **apostrophe**:

```
darth_vader = "No, I'm your father"

luke_skywalker = 'No! No! It\'s not true...'
```

Non-printing characters:

```
new_line = '\n'

a_tab = '\t'
```

Strings are Immutable

Strings are immutable, that is you **cannot change** it once you make it:

```
sample1 = 'spam'

sample1[1] = 'c' # ERROR!
```

However, you can **use it to make another string** (copy it, slice it, etc.)

```
sample1 = 'among us'

new_string = sample1[:-5] + 'g' + sample1[-2:]
```

Triple Quotes String

Triple quotes preserve both the **vertical** and **horizontal formatting** of the string

➔ Allows you to type tables, paragraphs, whatever and preserve the formatting

```
sample = '''this is

a test

Today'''
```

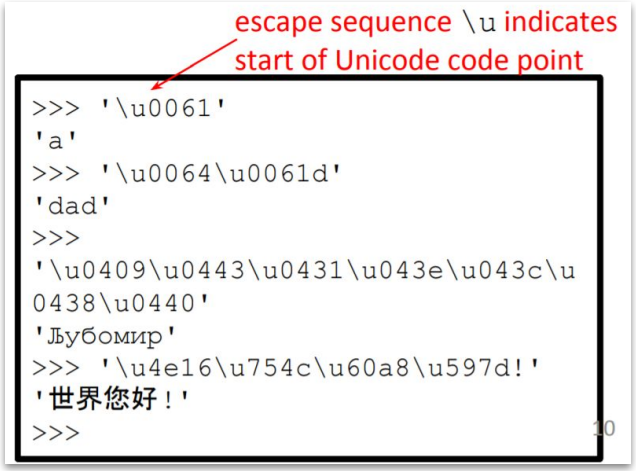
➔ This is also used for multi-line comments

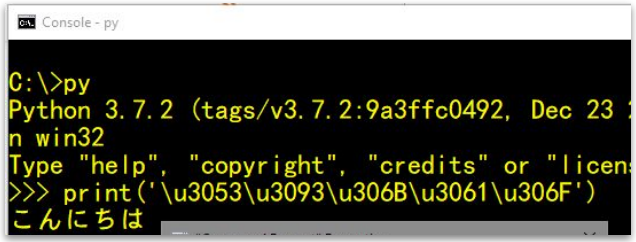
String Representation: Unicode (1)

A Unicode string is a **sequence of code points** that are numbers from **0 to 0x10FFFF**.

- ➔ The **code point** is not necessarily the actual byte representation of the character; it is just the **identifier** for the particular character.
- ➔ Unlike ASCII codes, Unicode code points are **not what is stored in memory**; the rule for translating a Unicode character or code point into a sequence of bytes is called an **encoding**.
- ➔ Unicode was designed so that, for any pair of characters from the same alphabet, the one that is **earlier** in the alphabet will have a **smaller Unicode code point**.

String Representation: Unicode (2)

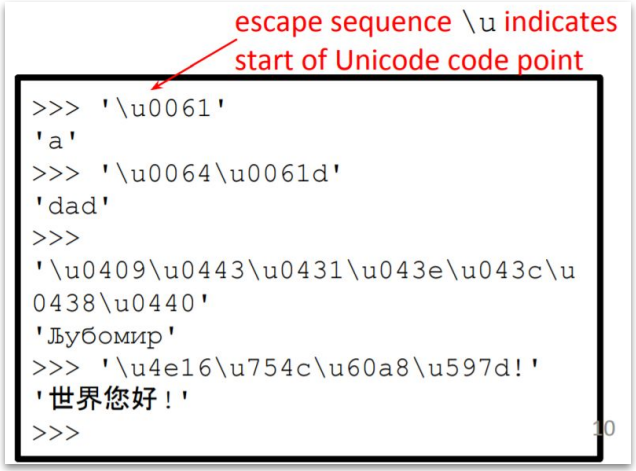


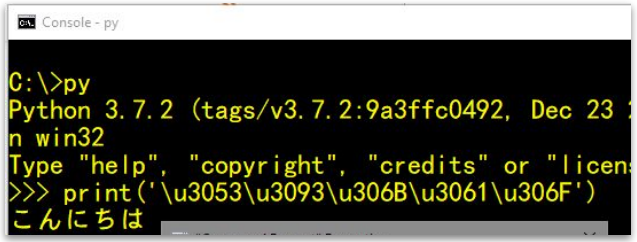


The code point for letter **a** is the integer with hexadecimal value **0x0061**

Unicode conveniently uses a code point for ASCII characters that is **equal to their ASCII code**

String Representation: Unicode (3)





There are several Unicode encodings: UTF-8, UTF-16, and UTF-32. UTF stands for **Unicode Transformation Format**.

- ➔ UTF-8 has become the preferred encoding for e-mail and web pages
- ➔ In UTF-8, every ASCII character has an encoding that is exactly the 8-bit ASCII encoding.

The default encoding when you write Python 3 programs is UTF-8.

String Representation: UTF-8

Every character is "mapped" (associated) with an **integer**

- ➔ UTF-8, a subset of Unicode, is such a mapping
- ➔ The function `ord()` takes a character and returns its UTF-8 **integer value**,
- ➔ The function `chr()` takes an integer and returns the UTF-8 **character**.

```
>>> ord('a')
97
>>> ord('?')
63
>>> ord('\n')
10
>>> chr(10)
'\n'
>>> chr(63)
'?'
>>> chr(97)
'a'
>>>
```

Subset of UTF-8

UTF-8 Complete Charset
https://www.fileformat.info/info/charset/UTF-8/list.htm

Char	Dec	Char	Dec	Char	Dec
SP	32	@	64	`	
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116

String Anatomy

Because the elements of a string are **a sequence**, we can associate **each element** with an **index**, a location in the sequence:

- ➔ positive values count up from the left, beginning with index 0
- ➔ negative values count down from the right, starting with -1

Characters:	M	I	D	O	R	I	Y	A		I	Z	U	K	U
Index (+):	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index (-):	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

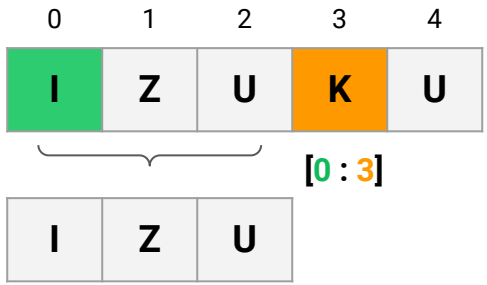
Accessing a String Element

A particular element of the string is **accessed by the index** of the element surrounded by square brackets[ ]

```
hello_str = 'Hello World'

print(hello_str[1]) # prints e

print(hello_str[-1]) # prints d

print(hello_str[-11]) # prints e

print(hello_str[11]) # ERROR
```

Slicing String: Basic Rule

Slicing is the ability to **select a subsequence** of the overall sequence



- ➔ uses the syntax **[start : finish]**, where:
  - start is the index of where we start the subsequence
  - finish is the index of a char after where we end the subsequence
- ➔ if either start or finish are not provided, it defaults to the beginning of the sequence for start and the end of the sequence for finish

`s[:m]` and `s[0:m]` are equivalent

Half Open Range for Slices

```
helloString = 'Hello World'
```

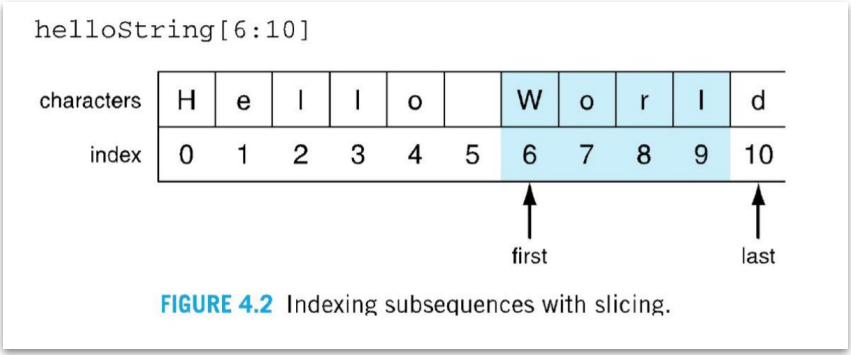


FIGURE 4.2 Indexing subsequences with slicing.

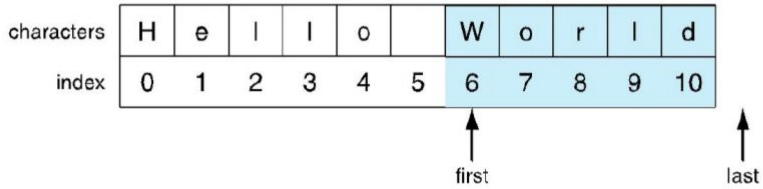
slicing uses what is called a half-open range

- ➔ the **first index is included** in the sequence
- ➔ the **last index is one after what is included**

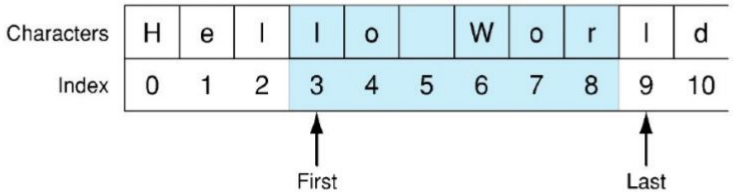
Slice Example

```
helloString = 'Hello World'
```

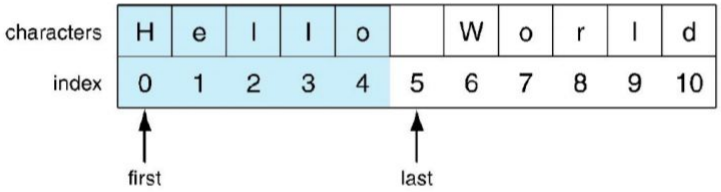
helloString[6:]



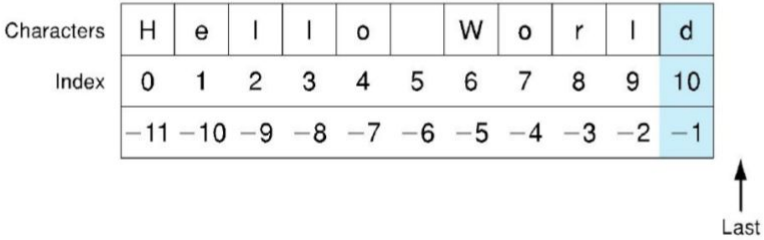
helloString[3:-2]



helloString[:5]



helloString[-1]



Extended Slicing

It also takes three arguments: `[start:finish:countBy]`

The defaults are: **Start is beginning, finish is end, countBy is 1**

```
my_str = 'hello world'
print(my_str[0:11:2]) # the expression is also equivalent to my_str[::2]

# the output is  'hlowrd'
```

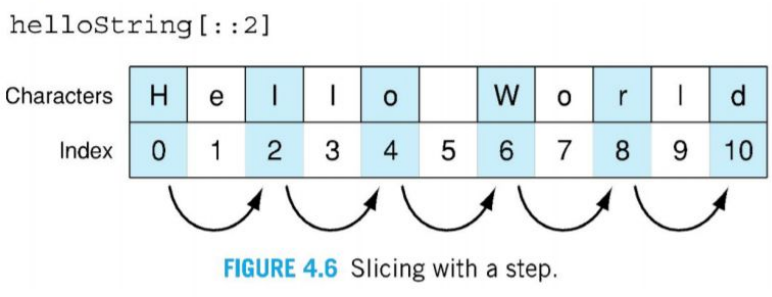


FIGURE 4.6 Slicing with a step.

Some Python Idioms

Idioms are python “phrases” that are used for a **common task** that might be less obvious to non-python folk

How to make a **copy** of a string:

```
my_str = 'hi mom'
new_str = my_str[:]
```

How to **reverse** a string:

```
my_str = "kasur ini rusak"
reverseStr = my_str[::-1]
```

Basic String Operations

```
old_str = 'muda-'
```

`+` is **concatenate**

```
new_str = 'muda' + '-' + old_str
print(new_str)
# the output is muda-muda-
```

`*` is **repeat**, the number is how many times the string is repeated

```
print(old_str * 10)
# the output is muda-muda-muda-muda-muda-muda-muda-muda-muda-muda-
```

Details on String Operations

Both `+` and `*` on strings **make a new string**, they do not modify the arguments

➔ **Order** of operation is important for **concatenation**, but it is irrelevant for repetition

```
new_str = 'str ke satu-' + 'str ke dua-' + 'str ke tiga-'
print(new_str) # the output is: str ke satu-str ke dua-str ke tiga-
```

➔ The types required are specific.
For **concatenation** you need **two strings**,
whereas for **repetition** you need a **string** and an **integer**

```
str_1 = 'one' + 'two'
str_2 = 'la'
print(str_1) # the output is: onetwo
print(str_2 * 10) # the output is: lalalalalalalalalala
print(10 * str_2) # the output is: lalalalalalalalalala
```

What does this program do?

Write the output in the **comment section**

```
sentence = 'okonomiyaki ga umai da'
# the length of the sentence is 22

mystery = sentence[1:5] + ' ' + sentence[-2] + sentence[-4] + sentence[0] + ' ' + sentence[20:22]

print(mystery)
```

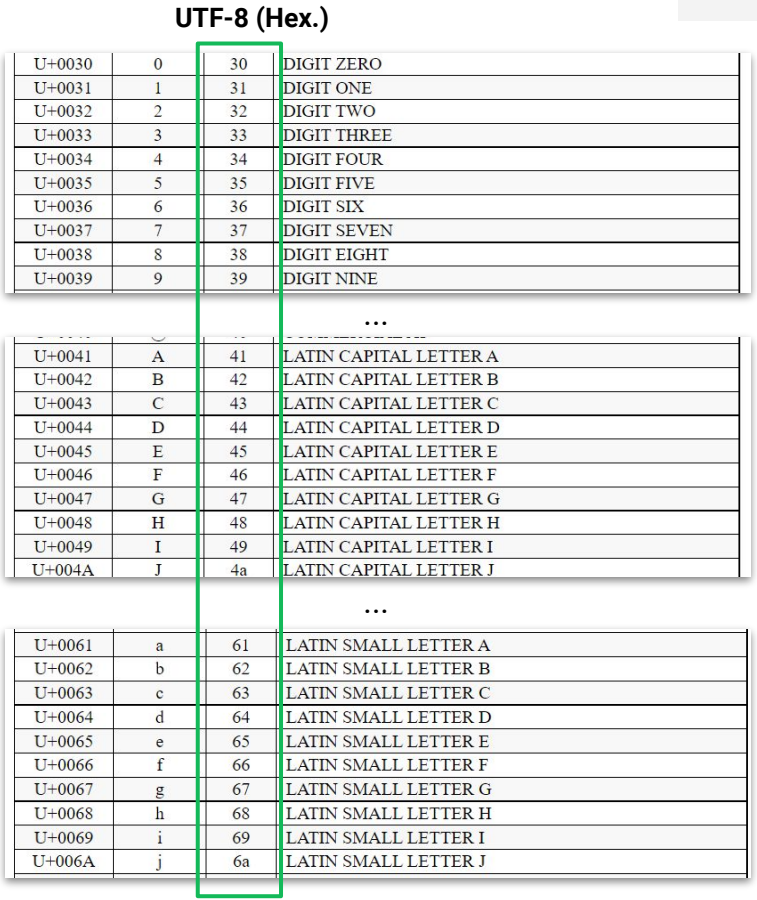
String Comparison: Single Char

Python 3 uses the Unicode mapping for characters.

Allows for representing **non-English characters**

UTF-8, subset of Unicode, takes the English letters, numbers and punctuation marks and **maps them to an integer**.

Single character comparisons are based on that number



U+0030	0	30	DIGIT ZERO
U+0031	1	31	DIGIT ONE
U+0032	2	32	DIGIT TWO
U+0033	3	33	DIGIT THREE
U+0034	4	34	DIGIT FOUR
U+0035	5	35	DIGIT FIVE
U+0036	6	36	DIGIT SIX
U+0037	7	37	DIGIT SEVEN
U+0038	8	38	DIGIT EIGHT
U+0039	9	39	DIGIT NINE

...

U+0041	A	41	LATIN CAPITAL LETTER A
U+0042	B	42	LATIN CAPITAL LETTER B
U+0043	C	43	LATIN CAPITAL LETTER C
U+0044	D	44	LATIN CAPITAL LETTER D
U+0045	E	45	LATIN CAPITAL LETTER E
U+0046	F	46	LATIN CAPITAL LETTER F
U+0047	G	47	LATIN CAPITAL LETTER G
U+0048	H	48	LATIN CAPITAL LETTER H
U+0049	I	49	LATIN CAPITAL LETTER I
U+004A	J	4a	LATIN CAPITAL LETTER J

...

U+0061	a	61	LATIN SMALL LETTER A
U+0062	b	62	LATIN SMALL LETTER B
U+0063	c	63	LATIN SMALL LETTER C
U+0064	d	64	LATIN SMALL LETTER D
U+0065	e	65	LATIN SMALL LETTER E
U+0066	f	66	LATIN SMALL LETTER F
U+0067	g	67	LATIN SMALL LETTER G
U+0068	h	68	LATIN SMALL LETTER H
U+0069	i	69	LATIN SMALL LETTER I
U+006A	j	6a	LATIN SMALL LETTER J

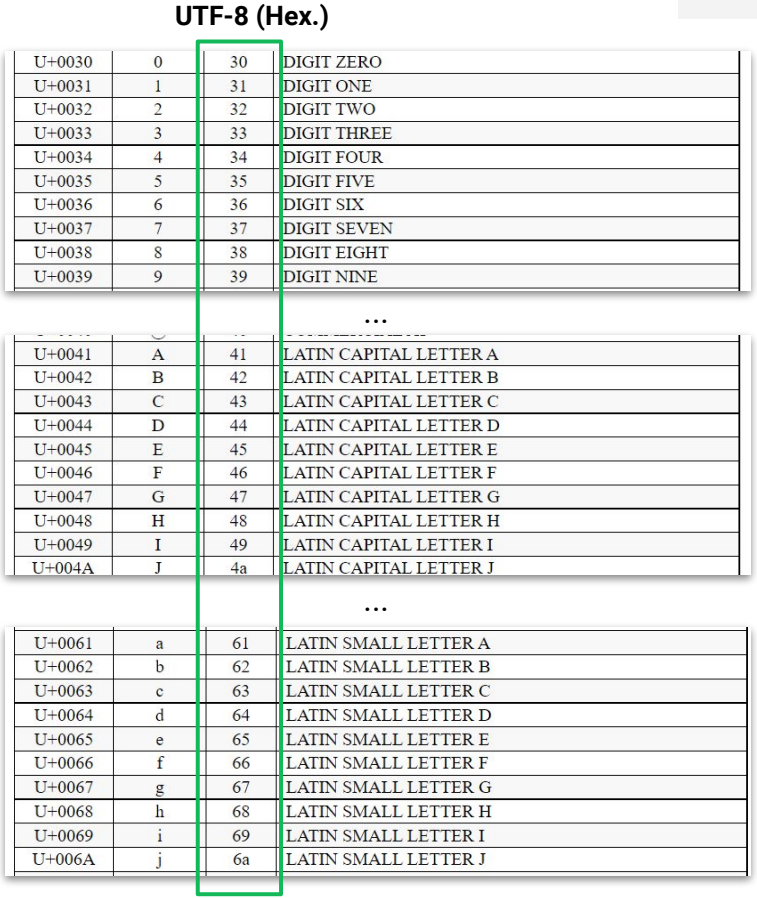
String Comparison within Sequence

It makes sense to **compare within a sequence** (lower case, upper case, digits)

'a' < 'b' □ True
'A' < 'B' □ True
'1' < '9' □ True

It can be weird outside of the sequence

'a' < 'A' □ False
'a' < '0' □ False



Whole Strings Comparison

Compare the **first element** of each string

- ➔ If they are equal, move on to the next character in each
- ➔ If they are not equal, the relationship between those to characters are the relationship between the string
- ➔ If one ends up being **shorter (but equal)**, the **shorter is smaller**

Examples

`'a' < 'b'` □ True

`'aaab' < 'aaac'` □ First difference is at the last char. `'b' < 'c'` so `'aaab'` is less than `'aaac'`. True

`'aa' < 'aaz'` □ The first string is the same but shorter. Thus it is smaller. True

Membership Operations

We can check to **see if a substring exists in the string** using the `in` operator.
It returns `True` or `False`

```
my_str = 'aabbccdd'
```

```
'a' in my_str □ True

'abb' in my_str □ True

'x' in my_str □ False
```

```
>>> string = 'Oikawa Tooru'
>>> 'Oi' in string
True
>>> 'oi' in string
False
>>> ' ' in string
True
```

Iterating String (1)

- To date we have seen the while loop as a way to iterate over a suite (a group of python statements)
- We briefly touched on the for statement for iteration, such as the elements of a list or a string
- We use the for statement to process each element of a list, one element at a time

```
for item in sequence:
    # suite
```

Iterating String (2)

```
my_str = 'abc'

for char in 'abc':
    print(char)
```

- ➔ first time through, `char = 'a' (my_str[0])`
- ➔ second time through, `char='b' (my_str[1])`
- ➔ third time through, `char='c' (my_str[2])`
- ➔ no more sequence left, for ends

What does this program do?

Write the output in the **comment section**

```
sentence = 'I, Giorno Giovanna have a dream'
word2 = 'g'
sums = 0

for letter in sentence:
     if letter == word2:
          sums += 1

print(sums)
```

1

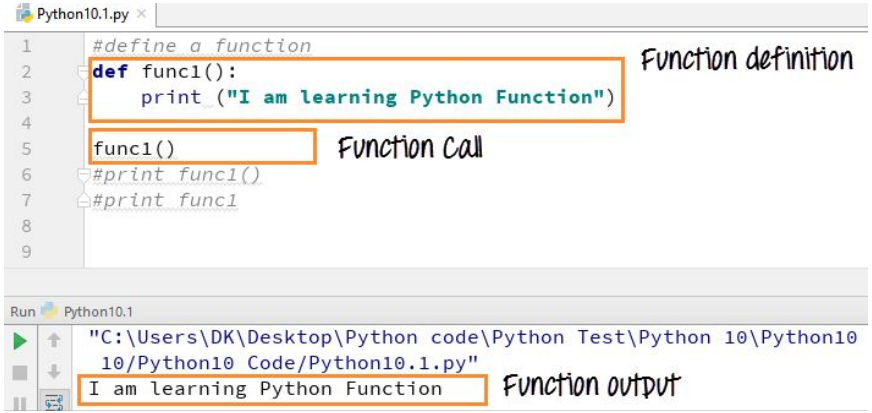
```
sentence = 'I, Giorno Giovanna have a dream'
word2 = 'g'
new_str = ''

for letter in sentence:
     new_str = new_str + letter +  '$'

print(new_str)
```

2

Function in a Nutshell

A function is a program that **performs some operation**. Its details are hidden (**encapsulated**), only it's interface provided.

A function takes some number of **inputs** (arguments) and returns a value based on the arguments and the function's operation.



String Function: `len()`

The len function takes as an argument a string and **returns an integer, the length of a string**.

```
my_str = 'Hello World'

length_of_my_str = len(my_str) # This will return 11. Space counts!
```

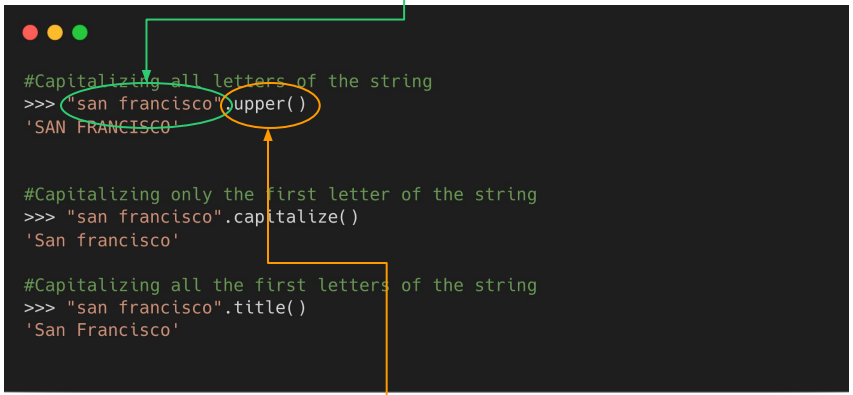
String Method

a method is a **variation on a function**

- ➔ like a function, it represents a **program**
- ➔ like a function, it has **input** arguments and an **output**

Unlike a function, it is applied **in the context of a particular object**. This is indicated by the **dot** notation invocation

This string (“San Fransisco”) is an **object**



This is method `upper()` applied on the string “San Fransisco”

More on Dot Notation

In general, dot notation looks like this.

```
object.method( … )
```

- ➔ It means that the **object** in front of the dot is **calling a method** that is associated with that object's type.
- ➔ The method's that can be called are tied to the type of the object calling it. Each type has different methods.

String Method Example: Upper

`upper()` is the name of a method. It generates a new string that has all **uppercase** characters of the string it was called with.

```
my_str = 'za warudo!'

print(my_str.upper()) # this will print 'ZA WARUDO!'
```

The `upper()` method was called in the context of `my_str`, indicated by the dot between them.

String Method Example: Find

```
my_str = 'hello'
my_str.find('l') # find index of 'l' in my_str ⇒ 2
```

Note how the method `find()` operates on the string object `my_str` and the two are associated by using the “dot” notation: `my_str.find('l')`.

Terminology: the thing(s) **in parenthesis**, i.e. the 'l' in this case, is called an **argument**.

String Method Example: Split

The split method will take a string and **break it into multiple new string parts** depending on the argument character.

By **default**, if no argument is provided, split is on any **whitespace** character (tab, blank, etc.)

You can assign the pieces with multiple assignment if you know how many pieces are yielded.

```
>>> string_1 = 'Hinata Shouyou'
>>> string_1.split()
['Hinata', 'Shouyou']
>>> string_2 = 'Nihongo Mantappu Jiwa'
>>> string_2.split()
['Nihongo', 'Mantappu', 'Jiwa']
>>>
```

```
>>> first_subs, second_subs = string_1.split()
>>> print(first_subs)
Hinata
>>> print(second_subs)
Shouyou
>>> first_subs, second_subs, third_subs = string_1.split()
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
>>>
```

Split Method Example: Reorder Name

```
name = 'Jean Pierre Polnareff'
first, middle, last = name.split()
transformed = last + ', ' + first + ' ' + middle

print(transformed) # the output is “Polnareff, Jean Pierre”
print(name) # the output is “Jean Pierre Polnareff”
print(first) # the output is “Jean”
print(middle)# the output is “Pierre”
```

Triggering Question 3

1. Create a program that can give an output of a string based on a user-inputted string which **its last character is capitalized.** The string is defined by the program user.
   Hint: use `upper()`

```
Masukkan string: apa
apA
```

```
Masukkan string: apa saja
apa sajA
```

2. Create a program to **reverse the word order in a '4 words' String.**
   The string is defined by the user and it always consists of 4 words separated by a space.

```
Masukkan string 4 kata: apa saja yang ingin
ingin yang saja apa
```

Share your codes in the **Scele Forum**

Chaining Method

Methods can be **chained** together.

1. Perform first operation, yielding an object
2. Use the yielded object for the next method

```
my_str = 'Python Rules!'
new_str_1 = my_str.upper() # This yields 'PYTHON RULES!'
new_str_2 = my_str.upper().find('O') # This yields 4, which is the location of letter 'O'
```

Optional Arguments

Some methods have **optional arguments**:

- ➔ If the user doesn't provide one of these, a default is assumed
- ➔ Find has a default second argument of 0, where the search begins

```
a_str = 'He had the bat'
var_1 = a_str.find('t')  # this yields 7, because the 1st 't',start at 0
var_2 = a_str.find('t',8) # this yields 13, the search start at 8, the 2nd 't' is at 13
```

Nesting Methods

You can “nest” methods, that is the result of one method as an argument to another (“**methods inside a method**”)

Remember that parenthetical expressions are done “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument

```
# translation: find the second 't'.
a_str = 'He had the bat'
var_1 = a_str.find('t', a_str.find('t') + 1)
```

More String Methods

Method	Description
strip()	removes any whitespace from the beginning or the end
lower()	Returns a string in lower case characters
upper()	Returns a string in uppercase characters
replace()	Replaces a string with another string
split()	Splits the string into sub strings
capitalize()	Capitalizes the first character in the string
count()	Returns no. of occurrences in the string
index()	Returns the index of the character
find()	Gives the index value of the string specified
isalpha()	Returns true if the string has only alphabets
isalnum()	Returns true if the string has both alphabets and numbers
isdigit()	Returns true if the string has only numbers
islower()	Returns true if the string has only lower case characters
isupper()	Returns true if the string has only uppercase characters

For more info:
https://docs.python.org/3/library/stdtypes.html#string-methods

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

Triggering Question 4

1. Create a program that can **capitalize certain character** within a string. The string and the character are defined by the user.

```
Masukkan kata: apa saja
Masukkan karakter yang ingin dikapitalisasi: a
Hasil akhir: ApA sAjA
```

2. Create a program that can **lower every capital letter** and **capitalize every lower letter** in a string (reversing the capitalization). The string is defined by the user.

```
Masukkan kata: ApA sAja
aPa SaJA
```

Share your codes in the **Scele Forum**

Review Questions

Explain what a string is and how we create it in Python.

Explain what Unicode is. Is it different from ASCII? Why?

Analyze the following code. What is the output?

```
s = 'Meliodas'
t = 'Hawk'
print('dasHa' in 2 * (s + t))
```

Analyze the following code. What is the output?

```
print(ord('Kamado Tanjirou'))
```

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Q&A Session