Python Basics

DA2303 – Python Programming

Table of contents

01

Basics

Comments, Literal Constants, Data Types, Variables, Statements

Basics

- Just printing hello world is not enough, is it?
- You want to do more than that - you want to take some input, manipulate it and get something out of it.
- We can achieve this in Python using constants and variables, and we'll learn some other concepts as well in this chapter.

Comments

- Comments are any text to the right of the # symbol and is mainly useful as notes for the reader of the program.
- For example:

```
print('hello world') # Note that print is a function
```

- or

```
# Note that print is a function
print('hello world')
```

Comments (cont.)

Useful Link:

Code tells you how, comments should tell you why.

- Use as many useful comments as you can in your program to:
  - explain assumptions
  - explain important decisions
  - explain important details
  - explain problems you're trying to solve
  - explain problems you're trying to overcome in your program, etc.

- This is useful for readers of your program so that they can easily understand what the program is doing.

- Remember, that person can be yourself after six months!

Literal Constants

- An example of a literal constant is a number like 5, 1.23, or a string like 'This is a string' or "It's a string!".
- It is called a literal because it is *literal* - you use its value literally.
- The number 2 always represents itself and nothing else - it is a *constant* because its value cannot be changed.
- Hence, all these are referred to as literal constants.

Numbers

- Numbers are mainly of two types - integers and floats.
- An example of an integer is 2 which is just a whole number.
- Examples of floating point numbers (or *floats* for short) are 3.23 and 52.3E-4.
- The E notation indicates powers of 10.
- In this case, 52.3E-4 means 52.3 * 10^-4^.

Boolean

- Boolean can only be two values – True or False
- Booleans are typically used in condition and comparison expressions that will be discussed further in the future.

Strings

- A string is a *sequence* of *characters*.
- Strings are basically just a bunch of words.
- You will be using strings in almost every Python program that you write.
- There are three (3) ways to specify strings.
  - Single Quote (')
  - Double Quotes (")
  - Tripe Quotes (""") or (''')

Single Quote

- You can specify strings using single quotes such as 'Quote me on this'.
- All white space i.e. spaces and tabs, within the quotes, are preserved as-is.

Double Quotes

- Strings in double quotes work exactly the same way as strings in single quotes. An example is "What's your name?".

Triple Quotes

- You can specify multi-line strings using triple quotes - (""" or '''). You can use single quotes and double quotes freely within the triple quotes. An example is:

```
'''This is a multi-line string. This is the first line.
This is the second line. "What's your name?," I asked.
He said "Bond, James Bond." '''
```

Strings Are Immutable

- This means that once you have created a string, you cannot change it.
- Although this might seem like a bad thing, it really isn’t.
- We will see why this is not a limitation in the various programs that we see later on.

The format method

- Sometimes we may want to construct strings from other information. This is where the format() method is useful.
- Save the following lines as a file str_format.py:

```
age = 20
name = 'Swaroop'

print('{0} was {1} years old when he wrote this book'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

The format method (cont.)

- Run the script through your terminal (or however you prefer):

```
$ python3 str_format.py
```

- Output:

```
$ python3 str_format.py
Swaroop was 20 years old when he wrote this book
Why is Swaroop playing with that python?
```

format(): How It Works (cont.)

- A string can use certain specifications and subsequently,
  the format method can be called to substitute those specifications with corresponding arguments to the format method.
- Observe the first usage where we use {0} and this corresponds to the variable name which is the first argument to the format method.
- Similarly, the second specification is {1} corresponding to age which is the second argument to the format method.

```
age = 20
name = 'Swaroop'

print('{0} was {1} years old when he wrote this book'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

Note that Python starts counting from 0 which means that first position is at index 0, second position is at index 1, and so on.

format(): How It Works (cont.)

- Notice that we could have achieved the same using string concatenation:

```
name + ' is ' + str(age) + ' years old'
```

- but that is much uglier and error-prone.
- Second, the conversion to string would be done automatically by the format method instead of the explicit conversion to strings needed in this case.
- Third, when using the format method, we can change the message without having to deal with the variables used and vice-versa.

format(): How It Works (cont.)

- Also note that the numbers are optional, so you could have also written as:

```
age = 20
name = 'Swaroop'

print('{} was {} years old when he wrote this book'.format(name, age))
print('Why is {} playing with that python?'.format(name))
```

- which will give the same exact output as the previous program.

format(): How It Works (cont.)

- We can also name the parameters:

```
age = 20
name = 'Swaroop'

print('{name} was {age} years old when he wrote this book'.format(name=name, age=age))
print('Why is {name} playing with that python?'.format(name=name))
```

- which will give the same exact output as the previous program.

format(): f-strings

- Python 3.6 introduced a shorter way to do named parameters, called "f-strings":

```
age = 20
name = 'Swaroop'

# notice the 'f' before the string
print(f'{name} was {age} years old when he wrote this book')
print(f'Why is {name} playing with that python?')
```

- which will give the same exact output as the previous program.

format(): detailed specs

- What Python does in the format method is that it substitutes each argument value into the place of the specification.
- There can be more detailed specifications such as:

```
# decimal (.) precision of 3 for float '0.333'
print('{0:.3f}'.format(1.0/3))

# fill with underscores (_) with the text centered
# (^) to 11 width '___hello___'
print('{0:_^11}'.format('hello'))

# keyword-based 'Swaroop wrote A Byte of Python'
print('{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python'))
```

format(): detailed specs (cont.)

```
# decimal (.) precision of 3 for float '0.333'
print('{0:.3f}'.format(1.0/3))

# fill with underscores (_) with the text centered
# (^) to 11 width '___hello___'
print('{0:_^11}'.format('hello'))

# keyword-based 'Swaroop wrote A Byte of Python'
print('{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python'))
```

Code from previous slide

- Output:

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

print(): new line

- Since we are discussing formatting, note that print always ends with an invisible "new line" character (\n) so that repeated calls to print will all print on a separate line each.

```
print('a')
print('b')
print('c')
```

output

```
a
b
c
```

print(): prevent new line

- To prevent this newline character from being printed, you can specify that it should:
  - end with a blank

```
print('a', end='')
print('b', end='')
```

output → `abc`

  - end with a space

```
print('a', end=' ')
print('b', end=' ')
print('c')
```

output → `a b c`

Escape Sequences

- Suppose, you want to have a string which contains a single quote ('), how will you specify this string?
- For example, the string is "What's your name?".
- You cannot specify 'What's your name?' because Python will be confused as to where the string starts and ends.
- So, you will have to specify that this single quote does not indicate the end of the string.
- This can be done with the help of what is called an *escape sequence*.

Escape Sequences (cont.)

- You specify the single quote as \' : notice the backslash.
- Now, you can specify the string as 'What\'s your name?’.
- Another way of specifying this specific string would be "What's your name?" i.e. using double quotes.
- Similarly, you have to use an escape sequence for using a double quote itself in a double quoted string.
- Also, you have to indicate the backslash itself using the escape sequence \\.

Escape Sequences: newline

- What if you wanted to specify a two-line string?
- One way is to use a triple-quoted string as shown previously
- or you can use an escape sequence for the newline character - \n to indicate the start of a new line. An example is:

```
'This is the first line\nThis is the second line'
```

Escape Sequences: tab

- Another useful escape sequence to know is the tab: \t.

```
'Next word is\ttabbed'
```

- There are many more escape sequences but I have mentioned only the most useful ones here.

Escape Sequences (cont.)

- One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added.
- For example:

```
"This is the first sentence. \
This is the second sentence."
```

- Is equivalent to:

```
"This is the first sentence. This is the second sentence."
```

Raw String

- If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a *raw* string by prefixing r or R to the string.

- An example is:

```
r"Newlines are indicated by \n"
```

Variable

- Using just literal constants can soon become boring - we need some way of storing any information and manipulate them as well.
- This is where *variables* come into the picture.
- Variables are exactly what the name implies - their value can vary, i.e., you can store anything using a variable.
- Variables are just parts of your computer's memory where you store some information.
- Unlike literal constants, you need some method of accessing these variables and hence you give them names.

Variable (cont.)

- Variables can hold values of different types called *data types*.
- The basic types are numbers, boolean and strings, which we have already discussed.
- In later chapters, we will see how to create our own types using classes.

Identifier Naming

- Variables are examples of identifiers.
- *Identifiers* are names given to identify *something*.
- There are some rules you have to follow for naming identifiers:
  - The first character of the identifier must be a letter of the alphabet (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore (_).
  - The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores (_) or digits (0-9).
  - Identifier names are case-sensitive. For example, myname and myName are *not* the same. Note the lowercase n in the former and the uppercase N in the latter.

Identifier Naming

- Examples of *valid* identifier names are
  - i
  - name_2_3.

- Examples of *invalid* identifier names are
  - 2things
  - this is spaced out
  - my-name
  - >a1b2_c3.

Example: var.py

- Example: Using variables and literal constants. Type and run the following program:

```
# Filename : var.py
i = 5
print(i)

i = 1.11
print(i)

b = True
print(b)

s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

output

```
5
1.11
True
This is a multi-line string.
This is the second line.
```

Example: var.py (How it works)

- First, we assign the literal constant value 5 to the variable i using the assignment operator (=). This line is called a statement because it states that something should be done and in this case, we connect the variable name i to the value 5.

- Next, we print the value of i using the print statement which, unsurprisingly, just prints the value of the variable to the screen.

- Then we assign another literal constant value 1.11 to the variable i. This will replace the value. We then print it and expectedly, we get the value 1.11.

- Similarly, we assign the boolean and literal string to the variables b and s respectively, and then print it.

Logical and Physical Line

- A physical line is what you *see* when you write the program.
- A logical line is what *Python sees* as a single statement.
- Python implicitly assumes that each *physical line* corresponds to a *logical line*.
- An example of a logical line is a statement like print('hello world') - if this was on a line by itself (as you see it in an editor), then this also corresponds to a physical line.
- Implicitly, Python encourages the use of a single statement per line which makes code more readable.

Logical and Physical Line (cont.)

- If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (;) which indicates the end of a logical line/statement.
- The following examples are effectively have the same two logical lines.

```
i = 5
print(i)
```

```
i = 5; print(i);
```

```
i = 5;
print(i);
```

```
i = 5; print(i)
```

Logical and Physical Line (cont.)

- However, I *strongly recommend* that you stick to *writing a maximum of a single logical line on each single physical line*.
- The idea is that you should never use the semicolon.
- In fact, most python programmers have *never* used or even seen a semicolon in a Python program.

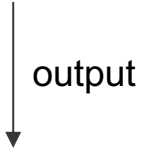
But I do sometimes accidentally added semicolon due to muscle memories doing Java programs.

Logical and Physical Line (cont.)

- There is one kind of situation where this concept is really useful: if you have a long line of code, you can break it into multiple physical lines by using the backslash.

- This is referred to as *explicit line joining*:

```
s = 'This is a string. \
This continues the string.'
print(s)
```




```
This is a string. This continues the string.
```

Logical and Physical Line (cont.)

- Similarly, these two (2) statements are the same.

```
i = \
5
```

```
i = 5
```

- Sometimes, there is an implicit assumption where you don't need to use a backslash.
- This is the case where the logical line has a starting parentheses, starting square brackets or a starting curly braces but not an ending one.
- This is called *implicit line joining*.
- You can see this in action when we write programs using list in later chapters.

Identation

- Whitespace is important in Python.
- Actually, *whitespace at the beginning of the line is important*.
- This is called *indentation*.
- Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

Identation (cont.)

- This means that statements which go together *must* have the same indentation.

```
i = 5
print('Value is', i)
print('I repeat, the value is', i)
```

- Each such set of statements is called a *block*.
- We will see examples of how blocks are important in later chapters.

Identation: error

- One thing you should remember is that wrong indentation can give rise to errors. For example:

```
i = 5
# Error below! Notice a single space at the start of the line
 print('Value is', i)
print('I repeat, the value is', i)
```

- When you run this, you get the following error:

```
  File "whitespace.py", line 3
    print('Value is', i)
    ^
IndentationError: unexpected indent
```

Identation: error (cont.)

- Notice that there is a single space at the beginning of the second line.
- The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written.
- What this means to you is that *you cannot arbitrarily start new blocks of statements* (except for the default main block which you have been using all along, of course).
- Cases where you can use new blocks will be detailed in later chapters.

How to indent

- Use four spaces for indentation.
- This is the official Python language recommendation.
- Good editors will automatically do this for you.
- Make sure you use a consistent number of spaces for indentation, otherwise your program will not run or will have unexpected behaviour.

02

Operators and Expressions

Arithmetic Operators,
Logical Operators,
Evaluation Order,
Expressions

Operators and Expressions

- Most statements (logical lines) that you write will contain *expressions*.
- A simple example of an expression is 2 + 3.
- An expression can be broken down into operators and operands.
- *Operators* are functionality that do something and can be represented by symbols such as + or by special keywords.
- Operators require some data to operate on and such data is called *operands*.
- In this case, 2 and 3 are the operands.

Operators

- We will briefly take a look at the operators and their usage.
- Note that you can evaluate the expressions given in the examples using the interpreter interactively.
- For example, to test the expression 2 + 3, use the interactive Python interpreter prompt:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Available Operators

- + (plus)
  - Adds two objects
  - 3 + 5 gives 8. 'a' + 'b' gives 'ab'.
- - (minus)
  - Gives the subtraction of one number from the other; if the first operand is absent it is assumed to be zero.
  - -5.2 gives a negative number and 50 - 24 gives 26.
- * (multiply)
  - Gives the multiplication of the two numbers or returns the string repeated that many times.
  - 2 * 3 gives 6. 'la' * 3 gives 'lalala'.

Available Operators (cont)

- ** (power)
  - Returns x to the power of y
  - 3 ** 4 gives 81 (i.e. 3 * 3 * 3 * 3)
- / (divide)
  - Divide x by y
  - 13 / 3 gives 4.333333333333333
- // (divide and floor)
  - Divide x by y and round the answer *down* to the nearest integer value. Note that if one of the values is a float, you'll get back a float.
  - 13 // 3 gives 4
  - -13 // 3 gives -5
  - 9//1.81 gives 4.0

Available Operators (cont)

- % (modulo)
  - Returns the remainder of the division
  - 13 % 3 gives 1. -25.5 % 2.25 gives 1.5.
- < (less than)
  - Returns whether x is less than y. All comparison operators return True or False. Note the capitalization of these names.
  - 5 < 3 gives False and 3 < 5 gives True.
  - Comparisons can be chained arbitrarily: 3 < 5 < 7 gives True.
- > (greater than)
  - Returns whether x is greater than y
  - 5 > 3 returns True. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False.

Available Operators (cont)

- <= (less than or equal to)
  - Returns whether x is less than or equal to y
  - x = 3; y = 6; x <= y returns True
- >= (greater than or equal to)
  - Returns whether x is greater than or equal to y
  - x = 4; y = 3; x >= 3 returns True
- == (equal to)
  - Compares if the objects are equal
  - x = 2; y = 2; x == y returns True
  - x = 'str'; y = 'stR'; x == y returns False
  - x = 'str'; y = 'str'; x == y returns True
- != (not equal to)
  - Compares if the objects are not equal
  - x = 2; y = 3; x != y returns True

Available Operators (cont)

- not (boolean NOT)
  - If x is True, it returns False. If x is False, it returns True.
  - x = True; not x returns False.
- and (boolean AND)
  - x and y returns False if x is False, else it returns evaluation of y
  - x = False; y = True; x and y returns False since x is False. In this case, Python will not evaluate y since it knows that the left hand side of the 'and' expression is False which implies that the whole expression will be False irrespective of the other values. This is called short-circuit evaluation.
- or (boolean OR)
  - If x is True, it returns True, else it returns evaluation of y
  - x = True; y = False; x or y returns True. Short-circuit evaluation applies here as well.

Available Operators (cont)

- More operators are available such as:
  - << (left shift)
  - >> (right shift)
  - & (bit-wise AND)
  - | (bit-wise OR)
  - ^ (bit-wise XOR)
  - ~ (bit-wise invert)

- We will not cover these. If you are interested, you may refer to the following link: https://wiki.python.org/moin/BitwiseOperators

Shortcut for math operation and assignment

- It is common to run a math operation on a variable and then assign the result of the operation back to the variable, hence there is a shortcut for such expressions:

```
a = 2
a = a * 3
```

- Can be written as:

```
a = 2
a *= 3
```

- Notice that var = var operation expression becomes var operation= expression.

Evaluation Order

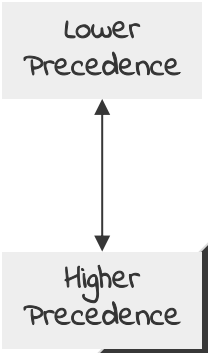
- If you had an expression such as 2 + 3 * 4, is the addition done first or the multiplication?
- Our high school maths tells us that the multiplication should be done first.
- This means that the multiplication operator has higher precedence than the addition operator.

Evaluation Order (cont.)

The operators which we have not already come across will be explained in the future (not the ones in red).

- The following table gives the precedence table for Python, from the lowest precedence (least binding) to the highest precedence (most binding).

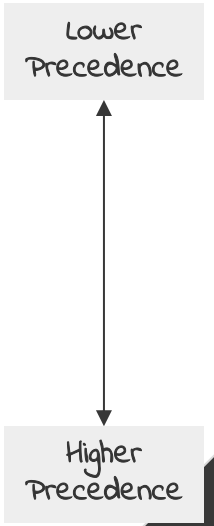
- This means that in a given expression, Python will first evaluate the operators and expressions lower in the table before the ones listed higher in the table.
    1. lambda : Lambda Expression
    2. if - else : Conditional expression
    3. or : Boolean OR
    4. and : Boolean AND
    5. not x : Boolean NOT
    6. in, not in, is, is not, <, <=, >, >=, !=, == : Comparisons, including membership tests and identity tests



Evaluation Order (cont.)

The operators which we have not already come across will be explained in the future (not the ones in red).

7. | : Bitwise OR
8. ^ : Bitwise XOR
9. & : Bitwise AND
10. <<, >> : Shifts
11. +, - : Addition and subtraction
12. *, /, //, % : Multiplication, Division, Floor Division and Remainder
13. +x, -x, ~x : Positive, Negative, bitwise NOT
14. ** : Exponentiation
15. x[index], x[index:index], x(arguments...), x.attribute : Subscription, slicing, call, attribute reference
16. (expressions...), [expressions...], {key: value...}, {expressions...} : Binding or tuple display, list display, dictionary display, set display



- Operators with the *same precedence* are listed in the same row in the above table. For example, + and - have the same precedence.

Changing the Order Of Evaluation

- To make the expressions more readable, we can use parentheses.
- For example, 2 + (3 * 4) is definitely easier to understand than 2 + 3 * 4 which requires knowledge of the operator precedences.
- As with everything else, the parentheses should be used reasonably (do not overdo it) and should not be redundant, as in (2 + (3 * 4)).
- There is an additional advantage to using parentheses - it helps us to change the order of evaluation.
- For example, if you want addition to be evaluated before multiplication in an expression, then you can write something like (2 + 3) * 4.

Associativity

- Operators are usually associated from left to right.
- This means that operators with the same precedence are evaluated in a left to right manner.
- For example, 2 + 3 + 4 is evaluated as (2 + 3) + 4.

Expressions

- Example (save as expression.py):

```
length = 5
breadth = 2
area = length * breadth

print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

- Output:

```
$ python expression.py
Area is 10
Perimeter is 14
```

Expression: How it works

- The length and breadth of the rectangle are stored in variables by the same name.
- We use these to calculate the area and perimeter of the rectangle with the help of expressions.
- We store the result of the expression length * breadth in the variable area and then print it using the print function.
- In the second case, we directly use the value of the expression 2 * (length + breadth) in the print function.

Expression: How it works (cont.)

- Also, notice how Python *pretty-prints* the output.
- Even though we have not specified a space between 'Area is' and the variable area, Python puts it for us so that we get a clean nice output and the program is much more readable this way (since we don't need to worry about spacing in the strings we use for output).
- This is an example of how Python makes life easy for the programmer.

03

String Operations

String methods, in operator, not in operator, String comparison

String operations

- A string is a **sequence** of characters.
- You can access the characters one at a time with the bracket operator:

```
fruit = 'banana'
letter = fruit[1]

print(letter)
```

- The second statement selects character number 1 from fruit and assigns it to letter.
- The expression in brackets is called an **index**.

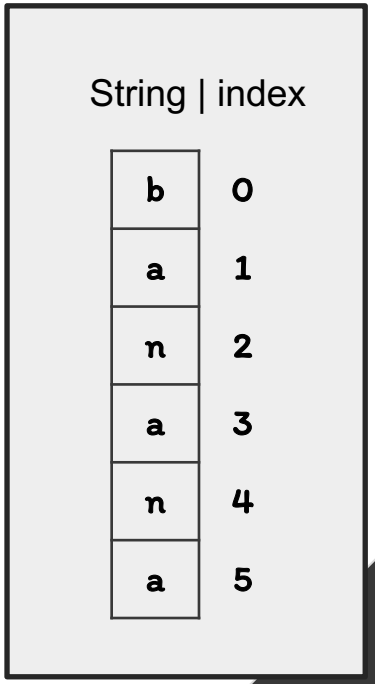
String operations (cont.)

- The index indicates which character in the sequence you want (hence the name).
- But the output you might not get what you expect: `a`
- For most people, the first letter of 'banana' is b, not a.
- But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

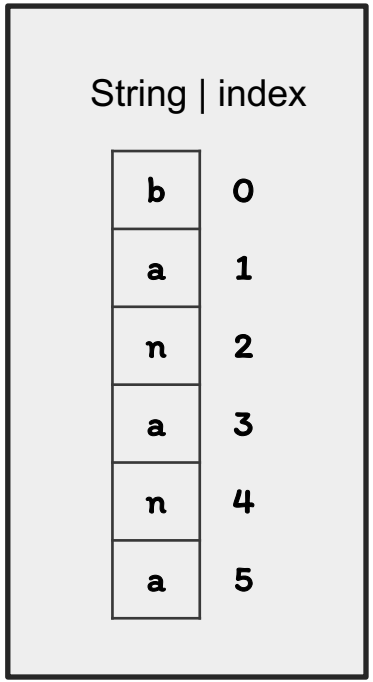
```
fruit = 'banana'
letter = fruit[0]

print(letter)
```

output → `b`



String operations (cont.)



- So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.
- You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer.
- Otherwise you get:

```
fruit = 'banana'
letter = fruit[1.5]

print(letter)
```

output

```
Traceback (most recent call last):
  File "fruit.py", line 2, in <module>
    letter = fruit[1.5]
TypeError: string indices must be integers
```

String operations: len

- len is a built-in function that returns the number of characters in a string:

```
fruit = 'banana'
length = len(fruit)

print(length)
```

- To get the last letter of a string, you might be tempted to try something like this:

```
fruit = 'banana'
length = len(fruit)
last = fruit[length]

print(last)
```

output

```
Traceback (most recent call last):
  File "last.py", line 3, in <module>
    last = fruit[length]
IndexError: string index out of range
```

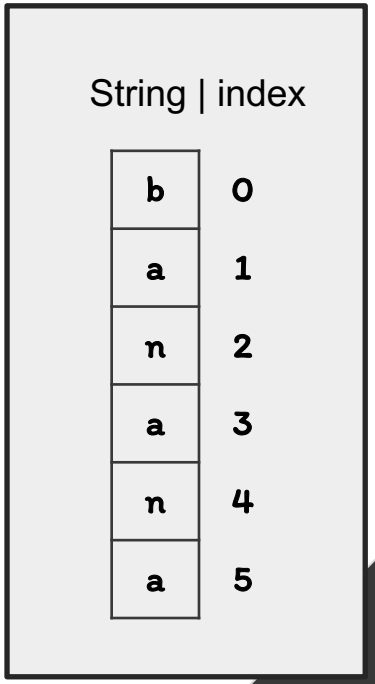
String operations: len (cont.)

- The reason for the IndexError is that there is no letter in ’banana’ with the index 6.
- Since we started counting at zero, the six letters are numbered 0 to 5.
- To get the last character, you have to subtract 1 from length:

```
fruit = 'banana’
length = len(fruit)
last = fruit[length-1]

print(last)
```

output → `a`



String operations: len (cont.)

- Alternatively, you can use negative indices, which count backward from the end of the string.
- The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

```
fruit = 'banana'
letter = fruit[-1]

print(letter)
```

output → `a`

```
fruit = 'banana'
letter = fruit[-2]

print(letter)
```

output → `n`

String operations: slices

- A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
s = 'Monty Python'

print(s[0:5])
print(s[6:12])
```

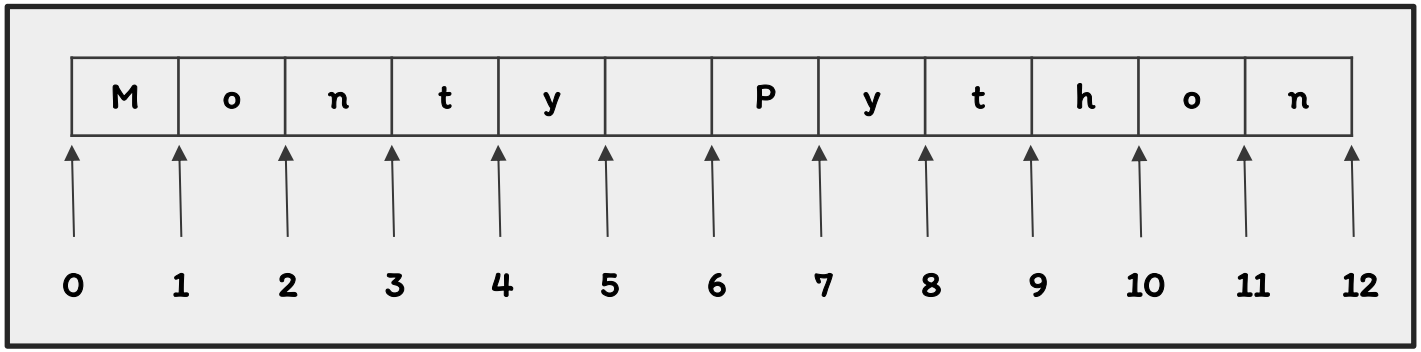
output

```
Monty
Python
```

- The operator [n:m] returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last.

String operations: slices (cont.)

- This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters.



String operations: slices (cont.)

- If you omit the first index (before the colon), the slice starts at the beginning of the string.
- If you omit the second index, the slice goes to the end of the string.

```
fruit = 'banana'

print(fruit[:3])
print(fruit[3:])
```

output

```
ban
ana
```

- If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

String operations: slices (cont.)

- If the first index is greater than or equal to the second the result is an **empty string**:

```
fruit = 'banana'

print(fruit[3:3])
```

output →

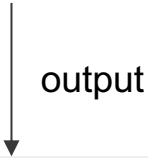
- An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Remember! Strings are immutable

- It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = 'Hello, world!'
greeting[0] = 'J'

print(greeting[0])
```




```
Traceback (most recent call last):
  File "immutable.py", line 2, in <module>
    greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

Strings are immutable (cont.)

- The “object” in this case is the string and the “item” is the character you tried to assign.
- For now, an **object** is the same thing as a value, but we will refine that definition later.
- An **item** is one of the values in a sequence.
- The reason for the error is that strings are **immutable**, which means you can’t change an existing string.

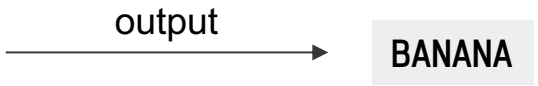
String methods

- A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different.

- For example, the method upper takes a string and returns a new string with all uppercase letters. Instead of the function syntax upper(word), it uses the method syntax word.upper():

```
word = 'banana'
new_word = word.upper()

print(new_word)
```



String methods (cont.)

- This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word.
- The empty parentheses indicate that this method takes no argument.

```
word.upper()
```

- A method call is called an **invocation**; in this case, we would say that we are invoking upper on the word.

String methods: find

- There is a string method named find, it takes a character and finds the index where that character first appears. If the character is not found, the function returns -1.

```
word = 'banana'
index = word.find('a')

print(index)
```

output → 1

- it can find substrings, not just characters:

```
word = 'banana'
index = word.find('na')

print(index)
```

output → 2

String methods: find (cont.)

- It can take as a second argument the index where it should start:

```
word = 'banana'
index = word.find('na', 3)

print(index)
```

output → 4

- And as a third argument the index where it should stop:

```
name = 'bob'
index = word.find('b', 1, 2)

print(index)
```

output → -1

This search fails
because b does not appear in
the index range
from 1 to 2 (not including 2).

String methods (cont.)

- If you refer to Python documentation in regards to String methods. It will state the find method as follows:

- str.find(*sub*[, *start*[, *end*]])
  - Return the lowest index in the string where substring *sub* is found within the slice s[start:end]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

- That means the argument surrounded with [] is optional. That means if you want to use the method, you don’t have to add argument for start and end.

- This is an important knowledge to know when reading the documentations.

More String methods

- str.count(*sub*[, *start*[, *end*]])
  - Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

- str.endswith(*suffix*[, *start*[, *end*]])
  - Return True if the string ends with the specified *suffix*, otherwise return False. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

- str.startswith(*prefix*[, *start*[, *end*]]
  - Return True if string starts with the *prefix*, otherwise return False. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

More String methods (cont.)

- str.isnumeric()
  - Return True if all characters in the string are numeric characters, and there is at least one character, False otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

- str.isupper()
  - Return True if all cased characters in the string are uppercase and there is at least one cased character, False otherwise.

- str.islower()
  - Return True if all cased characters in the string are lowercase and there is at least one cased character, False otherwise.

More String methods (cont.)

- str.lower()
  - Return a copy of the string with all the cased characters converted to lowercase.

- str.upper()
  - Return a copy of the string with all the cased characters converted to uppercase.

You can find more methods by refering to this Python documentations:

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

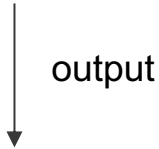
Please do try them.

Example: More String methods

- save as ds_str_methods.py

```
# This is a string object
name = 'Swaroop'

print('The string starts with "Swa"?', name.startswith('Swa'))
print('It contains the string "a"?', 'a' in name)
print('It contains the string "war"?', name.find('war') != -1)
```




```
The string starts with "Swa"? True
It contains the string "a"? True
It contains the string "war"? True
```

Example: More String methods – How it works

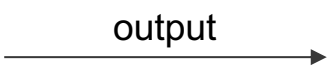
- Here, we see a lot of the string methods in action.
- The startswith method is used to find out whether the string starts with the given string.
- The in operator is used to check if a given string is a part of the string.
- The find method is used to locate the position of the given substring within the string; find returns -1 if it is unsuccessful in finding the substring.

The in operator

- The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
word = 'banana'
contain_a = 'a' in word
contain_seed = 'seed' in word

print(contain_a)
print(contain_seed)
```




```
True
False
```

The not in operator

- The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
word = 'banana'
contain_a = 'a' not in word
contain_seed = 'seed' not in word

print(contain_a)
print(contain_seed)
```

output

```
False
True
```

String comparison

- The logical operators work on strings. To see if two strings are equal (using == operator):

```
word = 'banana'
is_same = (word == 'banana')

print(is_same)
```

output → `True`

- Other logical operators (<, >, <= and >=) are useful for putting words in alphabetical order.

```
word = 'banana'
before_banana = ('apple' < word)

print(before_banana)
```

output → `True`

String comparison (cont.)

- Python does not handle uppercase and lowercase letters the same way that people do.
- All the uppercase letters come before all the lowercase letters, so:

```
word = 'banana'
before_banana = ('Durian' < word)

print(before_banana)
```

output → `True`

- A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

Thanks

Do you have any question?

jailani.rahman@pb.edu.bn
(+673) 223 4466 ext 241
Unit 6.01, Ong Sum Ping Campus,
Politeknik Brunei.