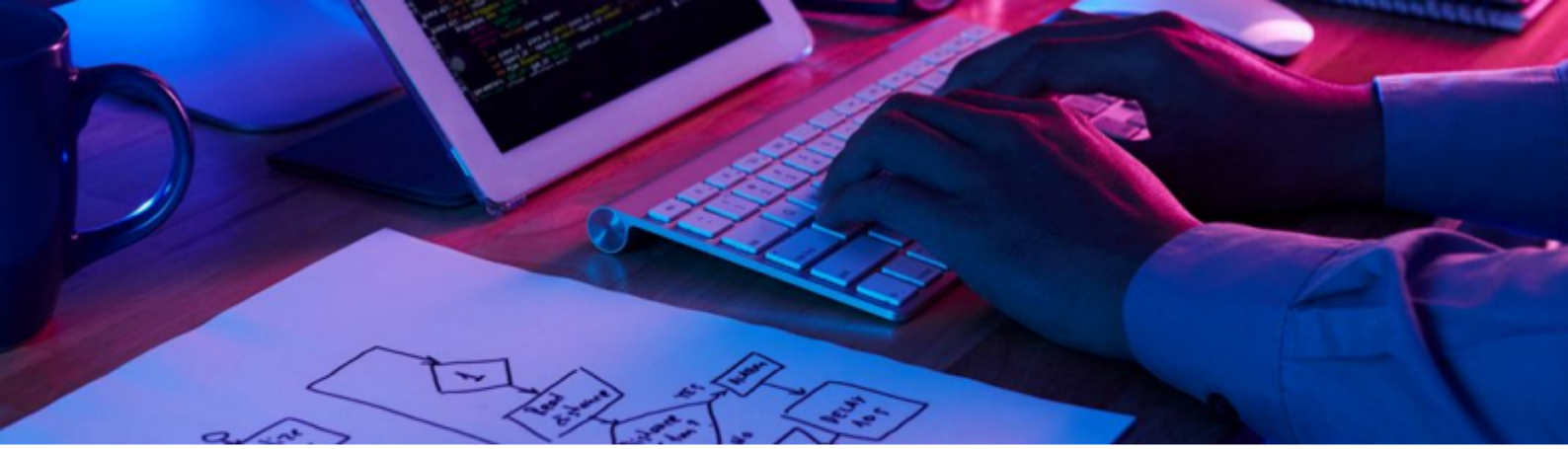


Crash Course on Python

Course 1 of 6 in Google IT Automation with Python Specialization.

bangkit

Programming with Python

Hello Python!

- Why programming with Python? Among many reasons is the easy syntax that's so convenient to both read and write that it feels.
- Python is one of the most chosen language for many of people working in IT (and technology in general).
- Python is omnipresent (available everywhere) on a wide variety of operating systems. Yet specifically in this course we'll more focus on Linux.

Hello in Python Programming Language

```
print('hello, bangkit!')

# this is comment,
# won't be interpreted
```

Basic Python Syntax: Data Types

- **String (str)**: text.
- **Integer (int)**: numbers, without fraction.
- **Float**: numbers with fraction.

We can convert from one data type to others by committing to implicit conversion or defining an explicit conversion.

Basic Python Syntax: Variables

- Variables are names that we give to certain values in our programs. Think of variables as containers for data.
- The values can be any data type.
- The process of storing a value inside a variable is called an assignment.
- Variable names can only be made up of letters, numbers, and underscore. Can't be Python reserved keywords.

Assign Value to Variables

```
length = 10
width = 2
area = length * width
print(area)
print(type(width))
print(type(str(area)))
```

Basic Python Syntax: Functions

- Define function with def keyword.
- Function can have a name used to later call the function using such name.
- After the name, function can optionally have parameters written between parenthesis. If it does, it can be one or more parameters.
- Function has body, written as a block after colon in function definition. The block has indented to the right, contains one/more statements.

Define Function, to print greeting

```
def greeting(name):
    print('hello, ' + name)
```

Indentation with 4 spaces

Call Function

```
greeting('bangkit')
```

Output

```
hello, bangkit
```

Basic Python Syntax: Functions with return value

- To get value from a function use the return keyword.
- Just like input values from parameters, the return values are optional. Therefore, with or without keyword return (incl. zero, one, or more values) is okay.
- Now with complete implementation (using input and output), the function can be easily used as the implementation of code reuse.

Returning Values from Function

```
def area_triangle(base, height):
    return base*height/2
```

Call Function and Save Return Value to Variable

```
area = area_triangle(10, 20)
print(area)
```

Basic Python Syntax: Comparison

- **Boolean (bool)** data type represents one of two possible states, either True or False.
- Not all data types can be compared, so be aware to compare two different data types.
- Comparison operator not only checking equality and less/more it also includes logical operator: **and**, **or**, **not**.

Expressions of Comparison

```
print(1 < 10)
True

print("Linux" == "Windows")
False

print(1 != "1")
True

print(not True)
False
```

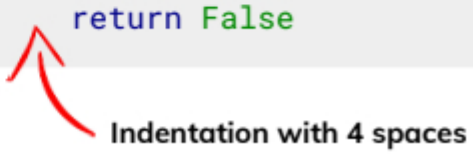
Basic Python Syntax: Conditionals

- The ability of a program to alter its execution sequence is called branching.
- To branch an execution, only if it matches a certain condition, use if keyword. The if block will be executed **only if the condition is True**.
- Sometimes the conditions are binary, so if it's not match one condition the other condition use else keyword.

Condition Evaluations

```
if hour < 12:
    print("Good morning!")


def is_negative(number):
    if number < 0:
        return True
    else:
        return False
```

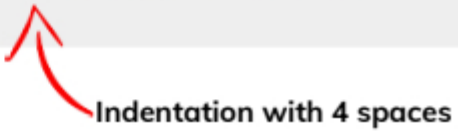


Basic Python Syntax: Branching Multiple Conditions

- Occasionally, there will be multiple conditions to check, this is where the elif statement, which is **short for else if**, comes into play.
- Just like else keyword, elif keyword is only usable if there's an if keyword. The difference is that elif keyword can be used multiple times.

Multiple Conditions Evaluation

```
def check(number):
    if number > 0:
        return "Positive"
    elif number == 0:
        return "Zero"
    else:
        return "Negative"
```



Loops: while

- while loop instruct computer to continuously execute code based on the value of a condition.
- Just like conditions, the while loop body has its own block, indented to the right.
- The statements inside the while loop block will be repeated as long as the condition value still True.

while loop

```
x = 7  # also try with x = 0

while x > 0:
    print("positive x=" + str(x))
    x = x - 1
print("now x=" + str(x))
```

Initialization of variable

The conditions

Pay attention to the indentation difference, outside the loop block

Loops: for

- for loop iterates over a sequence of values.
- One simple example iterates over a sequence of numbers, which is generated by function **range**.
- Also take a look at in keyword, that separates between the sequence and variable that will take each of the values in the sequences and used in the iteration block.

for loop

```
for x in range(3):  # 0, 1, 2
    print("x=" + str(x))

for x in range(3, 0, -1):
    # 3, 2, 1
    print(x)
```

Loops: break & continue

- Both while and for loops can be interrupted using the break keyword. Normally do this to interrupt a cycle due to a separate condition.
- In other occasion, use the continue keyword to skip the current iteration and continue with the next one. It is typically used to jump ahead when some of the elements of the sequences are not relevant.

break from loop

```
for x in range(3):
    print("x=" + str(x))
    if x == 1:
        break  # quit from loop
```

continue inside loop

```
for x in range(3, 0, -1):
    if x % 2 == 0:
        continue  # skip even
    print(x)
```

Nested for Loops

- There will be some occasions to write for loop inside a for loop, which is called nested for loops.
- One simple example, using nested for loops to find prime numbers.

Note: the example focus on code simplicity, not focus on optimization.

Nested for Loops

```
for i in range(2, 10):  # 2-9
    is_prime = True
    for j in range(2, i):
        if i % j == 0:
            is_prime = False
            break
    if is_prime:
        print(str(i) + " is prime ")
```

Output

```
2 is prime
3 is prime
5 is prime
7 is prime
```

Recursion (Optional)

- **A**: text.
- **B**: text.

Strings

- String is a data type in Python employed to represent a piece of text. It's written between quotes, either single quotes, double quotes, or triple quotes. Escape character using backslash (\).
- String can be as short as zero characters (empty string) or significantly long. String concatenation using plus sign (+). The **len** function tells the number of characters contained in the string.

Strings

```
program_name = 'bangkit'
program_year = "it's the 2nd"
multi_line = """hello,
email test.
signature."""

# let's
# "bangkit"
print("let's\n\""+program_name+"\"")

print(len(''))  # 0
print(len(program_name)==7)  # True
```

String Indexing and Slicing

- Python starts counting indexes from 0 not 1. Access index greater than its length - 1, triggers index out of range. Negative indexes starts from behind.
- To access substring, use slicing, similar to index, with range using a colon as a separator, starts from first number, up to 1 less than last.
- Slicing with one of two indexes means the other index is either 0 for the first value or its length for the second value.

String Indexing and Slicing

```
name = 'bangkit'
print(name[1])  # a
print(name[len(name)-1])  # t
print(name[-1])  # t
print(name[-2])  # i

print(name[4:len(name)-1])  # ki

print(name[:4])  # bang (0-3)
print(name[4:])  # kit  (4-len)
```

Strings are immutable

- Strings in Python are immutable, meaning they can't be modified, can’t change individual characters. It'll trigger TypeError object does not support item assignment.
- To change string, replace it with the new string.
- Use in keyword to check if substring is a part of the string.

Immutable String

```
year = "it's 2021"
year[-1] = "0"  # TypeError

year = year[:-1] + "0"
print(year)  # it's 2020

print('2020' in year)  # True
```

String Methods

- string class provide a bunch of methods for working with text. Not only related to text modification, there's also many of text checking method.
- Remember, the goal is not for memorize all of the methods, just check the documentation or search on the web anytime.

String Methods

```
program = 'bangkit 2021'
print(program.index('g'))  # 3

print(program.upper())  # BANGKIT 2021
print(program.endswith('2021'))  # True
print(program.replace('2021', '2020'))

year = 2021  # integer 2021
print(str(year).isnumeric())  # True
# bangkit for 2021
print("{} for {}".format("bangkit",
year))
```

* Complete string methods in Python docs https://docs.python.org/3/library/stdtypes.html#string-methods

Lists

- Think of list as container with space inside divided up into different slots. Each slot can contain a different value.
- Python use square brackets [] to indicate where the list starts and ends. list indexes starts from 0, just like string, also slicing to return another list.

Lists are mutable

- If strings are immutable, lists are mutable, means able to add, remove, or modify elements in a list.
- Use **append** to add to last element. To add on specific index, use **insert**. To delete element, use **remove** with element or **pop** with index.
- For element modification, change directly to the specific index.

Mutable list

```
paths = ['ML', 'Cloud']
paths.append('Android')
print(len(paths))  # 3
paths.remove('Android')
paths.insert(1, 'Mobile')

# ['ML', 'Mobile', 'Cloud']
print(paths)
paths.append('Python')
paths.pop(-1)  # remove 'Python'

# change 'ML' to 'Machine Learning'
paths[0] = 'Machine Learning'
```

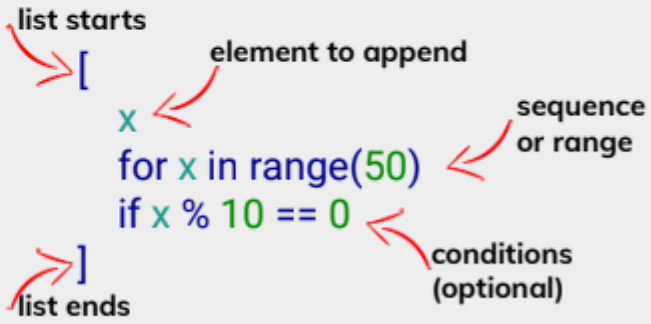
List Comprehensions

- Create a new list from a sequence or a range in single line using list comprehensions.
- List comprehensions can be really powerful, but can also be utterly complex, resulting to codes that are hard to read.

list Comprehensions

```
even = [x*2 for x in range(1,5)]
print(even)  # [2, 4, 6, 8]

tens = [x for x in range(50) if x % 10 == 0]
print(tens)  # [0, 10, 20, 30, 40]
```



Tuples

- Tuples are like lists. They can contain elements of any data type. But, unlike lists, tuples are immutable.
- Python using parentheses () to indicate where the tuple starts and ends.
- Good example of tuple is when a function returns multiple values.

Strings, Lists, and Tuples are included as sequence types.

Tuples

```
def get_stat(numbers):
  total = sum(numbers)
  length = len(numbers)
  mean = total / length
  return length, total, mean

stat = get_stat([1, 3, 5, 7])
print(stat)  # (4, 16, 4.0)
print(type(stat))  # tuple

for data in stat:
    print(data)  # element per line
```

Dictionaries

- Like lists, dictionaries are used to organize elements into collections. Unlike lists, not accessing elements inside dictionaries using position.
- Data inside dictionaries take the form of **pairs of keys and values**. To get a dictionary value, use its corresponding key.
- Not like list index must be a number, type of key in dictionary use strings, integers, tuples & more.
- dictionary use curly brackets {}.

Dictionaries

```
students = {
    'ml': 500,
    'mobile': 700,
    'cloud': 900
}
print(type(students))     # dict
print(students['cloud'])  # 900

# keys: ['ml', 'mobile', 'cloud']
for key in students.keys():
    # eg: ml:500
    print(key + ':' + students[key])
```

Iterating Over Dictionaries

- Use for loops to iterate through the contents of dictionary (implicitly over keys).
- To get both key and value as tuple at the same time, use **items**.
- Other than using **keys** to get all keys, use **values** to get all dictionary values.

dictionary Iterations

```
file_counts = {"jpg": 10,
               "txt": 14,
               "csv": 2,
               "py": 23}
for extension in file_counts:
    print(extension)  # eg: jpg

for ext, amount in file_counts.items():
    print('{} files .{}'
          .format(amount, ext))
```

Dictionaries are mutable

- Just like lists, dictionaries are mutable, means able to add, remove, or modify elements in a dictionary.
- Set new value using associated key. Add item (pairs of key & value) by set new key with new value. Delete item with **del** keyword or delete all items using **clear**.

Use dictionary over list if aims to access data via its key instead of iterate to find the key.

Mutable dictionary

```
# point in line y = 2x + 1
point_a = {'x': 2, 'y': 5}
point_a['x'] = 3
point_a['y'] = 7

new_point = {}  # empty dictionary
new_point['z'] = 2
print(len(new_point.keys()))  # 1
del new_point['z']  # remove item
print(new_point)    # {}

new_point = {'x': 0, 'y': 1}
new_point.clear()
print(new_point)    # {}
```

Object Oriented Programming (Optional)

- Python uses a programming pattern called object-oriented programming, which models concepts using **classes** & **objects**. This is a flexible and powerful paradigm where classes represent and define concepts, while objects are instances of classes.
- The idea of object-oriented programming might sound abstract and complex, but you've actually been using objects already without even realize it. Almost **everything in Python is an object**, all of the numbers, strings, lists, and dictionaries we've seen so far have been objects.

Object Oriented Programming (Optional)

- The core concept of object-oriented programming comes down to attributes and methods associated with a type. The attributes are the characteristics associated to a type and the methods are the functions associated to a type.
- Let's think about an IT focused **example**, like a **file** in a computer. It has many **attributes**, it has a name, a size, the date it was created and a whole lot more. The typical file object focuses on the file's contents, and so this object has various **methods** to read and modify what's inside the file.

Object Oriented Programming (Optional)

- To create and define classes in Python similar to define functions. It starts with the **class** keyword, followed by the class name and a colon. Python style guide recommends class names to start with capital letter.
- After the class definition line is the class body, indented to the right. We can define attributes for the class inside the class body.

Defining New Classes

```
>>> class Flower:
...     pass
...
>>>
>>> class Apple:
...     color = ""
...     flavor = ""
...
>>>
```

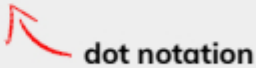
most simple class definition

Object Oriented Programming (Optional)

- Create a new instance of the new class by assigning it to a variable. This is done by calling the class name as if it were a function.
- Set the attributes of the class instance by accessing them using **dot notation**. It can be used to set or retrieve object attributes, as well as call methods associated with the class.

Defining Classes

```
>>> class Apple:
...     color = ""
...     flavor = ""
...
>>> golden = Apple()
>>> golden.color = "Yellow"
>>> golden.flavor = "Soft"
>>>
```



Classes and Methods (Optional)

- **Methods** are functions that operate on the **attributes** of a specific instance of a **class**.
- To define a method use the **def** keyword, just like a function. A function as a method of the class is receiving a parameter called **self**. It represents the instance of the method is being executed on.

Defining Class' Method

```
>>> class Piglet:
...     def speak(self):
...         print("oink oink")
...
>>> hamlet = Piglet()
>>> hamlet.speak()
oink oink
>>>
```

Classes and Methods (Optional)

- All methods that start and end with two underscores are special methods. **Constructor** is one of very important special methods. It's always named `__init__`.
- To get the string representation of an object, use method called `__str__`.

Defining Constructor and Special Methods

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "{} apple"\
...                .format(self.color)
...
>>> golden = Apple("Yellow", "Soft")
>>> print(golden.color)
Yellow
>>> print(golden)
Yellow apple
>>>
```


bangkit