Python Boot Camp

HKU Business School
The University of Hong Kong

Instructor: Dr. DING Chao

Agenda

- Why Python
- Install and Run Python
- The Very Basics of Python
  - Variables, expressions, and statements
  - Conditional execution
  - Functions
  - Modules and packages
  - Loops and iterations
  - Strings
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - File I/O
  - One Liners
  - Object-Oriented Programming (OOP)

Why Python

Why Python?

- Created in 1989 by Guido van Rossum.
- Python was ranked the most popular programing language by IEEE Spectrum in the last few years in a roll.

✓Object-oriented
✓Open source
✓Support libraries
✓Ease of use
✓Component integration
✓Enjoyment
✓IDEAL for data science!
✓Extremely powerful

Rank	Language	Type	Score
1	Python		100.0
2	Java		95.3
3	C		94.6
4	C++		87.0
5	JavaScript		79.5
6	R		78.6
7	Arduino		73.2
8	Go		73.1
9	Swift		70.5
10	Matlab		68.4

Source: IEEE Spectrum

Who uses Python?

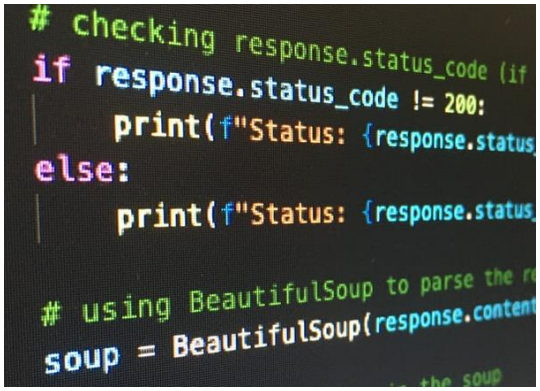
- Web services
- Hardware testing
- Financial market forecasting
- Movie/Game development
- Scientific computing

More details on https://www.python.org/about/success/

Install and Run Python

How to use Python?

+



The core of the Python language
https://www.python.org/

An environment to edit and execute your Python codes

Integrated Development Environment

- An integrated development environment (IDE) is software for building applications that combines common developer tools into a single graphical user interface (GUI).
- Due to Python's open-source nature, it has numerous IDEs.

Our Choice – Colab

- Google's Colaboratory (Colab) offers a cloud-based Jupyter Notebook environment.
- Free access at https://colab.research.google.com/
- No installation needed at all.
- Python scripts are stored on your Google Drive.
- It uses the computing power of a remote server, not your local computer.

Variables, Expressions, and Statements

Constants and Variables

- Fixed values such as numbers, letters, and strings, are called constants.
- We generally save constant values in variables for later use by calling the variable name.
- We get to choose the names of the variables.
- We can also change the contents of a variable in a later statement.

```
>>> course_code = 'Python 101'
>>> _CGPA = 3.53
>>> pi = 3.1415926535897931
```

Note: >>> indicates input.

Variable Naming Rules

1. Must start with a letter or underscore _
2. Must consist of letters, numbers, or underscores
3. Case Sensitive
4. Do not use reserved words as variable names.

```
False    class    return   is       finally
None     if       for lambda   continue
True     def from     while    nonlocal
and del global   not with
as   elif     try      or       yield
assert   else     import   pass
break    except   in       raise
```

- Recommended practices when naming variables:
  - ✓ Meaningful and succinct
  - ✓ Lowercase, concatenate words with underscore: my_very_long_name
  - ✓ Or, use camelCase: myVeryLongName

Assignment Statements

- We assign a constant value or an expression to a variable using the assignment statement (=).

```
x = 3.9
y = 'python'
_z = 3.9 * x - 9
```

- Python evaluates the right side before it assigns the value to the left. Therefore, it is ok to update a variable in place.

```
>>> temp = 23
>>> temp = temp + 1
```

- If you update a variable that doesn't exist, you get an error.

```
>>> k = salary + 1
NameError: name 'salary' is not defined
```

Arithmetic Operators

- Double asterisks is exponentiation (raise to a power).
- % and // are called modulus operators.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder
//	Quotient

Shortcut	Description
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

Order of Evaluation

- When we string operators together - Python must know which one to do first.
- This is called "operator precedence".

```
x = 1 + 2 * 3 - 4 / 5 ** 6
```

Parenthesis
Power
Multiplication
Addition
Left to Right

Comments in Python

- Anything after a # (hashtag) is ignored by Python.
- Why comment?
  - Describe what is going to happen in a sequence of code
  - Document who wrote the code or other ancillary information
  - Turn off a line of code - perhaps temporarily

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60

percentage = (minute * 100) / 60 # percentage of an hour

v = 5 # assign 5 to v, added by Chao on Jan 21st, 2021

s = 10 # velocity in meters/second.

# something needs to happen here, come back later.
```

Basic Data Types in Python

- Numbers
  - Integers: x = 2, y = 5
  - Floats: x = 3.456, y = 9823.9
- Strings
  - my_name = 'Chao'
  - my_color = 'Green'
- Lists: e.g., my_lst = [1, 2, 3, 'number']
- Tuples: e.g., my_tpl = (1, 2, 3, 'abc')
- Sets: e.g., my_set = {6, -9, 12, 8}
- Dictionaries: e.g., my_dict = {'name': 'Eric', 'age': 18}

Corresponding keywords

```
int
float
str
list
tuple
set
dict
```

Type Matters

- Python knows what “type” everything is. Some operations are prohibited, e.g., “add 1” to a string.

```
>>> eee = 'hello ' + 1

TypeError: Can't convert 'int'
object to str implicitly
```

- We find what type something is by using the **`type`** function.

```
>>> type(1)
<class 'int'>

>>> type(98.6)
<class 'float'>

>>> type('hku')
<class 'str'>
```

Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float.

```
>>> 99.0 + 100
199.0
```

- Use **`int`** and **`float`** to convert between integers and floats.

```
>>> int(42.6)
42
```

- Use **`int`** and **`str`** to convert between integers and strings.

```
>>> str(42)
'42'
```

Errors and Exceptions

- Syntax: set of rules to be followed when writing program.
- Syntax error: an error that makes it impossible to parse and thus impossible to interpret.

```
20years
9*/4
```

SyntaxError: invalid syntax

- Logical error, also called exception: an error in a program that makes it do something other than what the programmer intended while the syntax is totally fine.

```
grades = 95
a = grades / 0
print(a)
```

ZeroDivisionError: division by zero

Errors and Exceptions

Common Exceptions

Exception	Description
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when the assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program runs out of memory.
TypeError	It occurs when a function and operation are applied in an incorrect type.

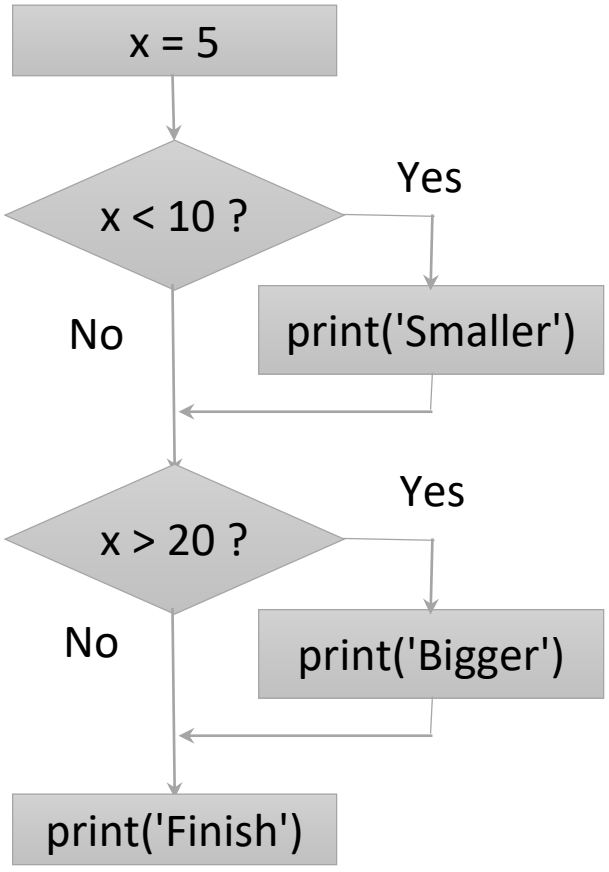
The Most Common Rookie Mistake

TYPOS

- Tips:
  - Use intuitive names
  - Use autocomplete
  - Check ([{ ' are in pairs

Conditional Execution

Conditional Steps




```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')
print('Finish')
```

Smaller
Finish

Comparison Operators

- Boolean expressions ask a question and produce a True or False result which we use to control program flow.
- Boolean expressions use comparison operators to evaluate conditions.
- Comparison operators do not change the variables.

Operator	Description
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal
is	Equal to in term of identity

Remember: "=" is used for assignment.

Logical Operators

- Logical operators are used to combine multiple conditions together and evaluate them as a single Boolean expression.

Operator	Description	Example
`and (&)`	True if both the operands are True	a and b
`or (|)`	True if either of the operands is True	a or b
`not`	True if the operand is False	not a

The Most Common Rookie Mistakes

1. Forgot the colon (“:”) at the end of the condition.
2. Incorrect indentation.

```
if age >= 65
    print('……')
```

```
if age>=65:
print('……')
```

- Tips:
  - Missing colon:

    SyntaxError: invalid syntax

  - Wrong indentation:

    IndentationError: expected an indented block

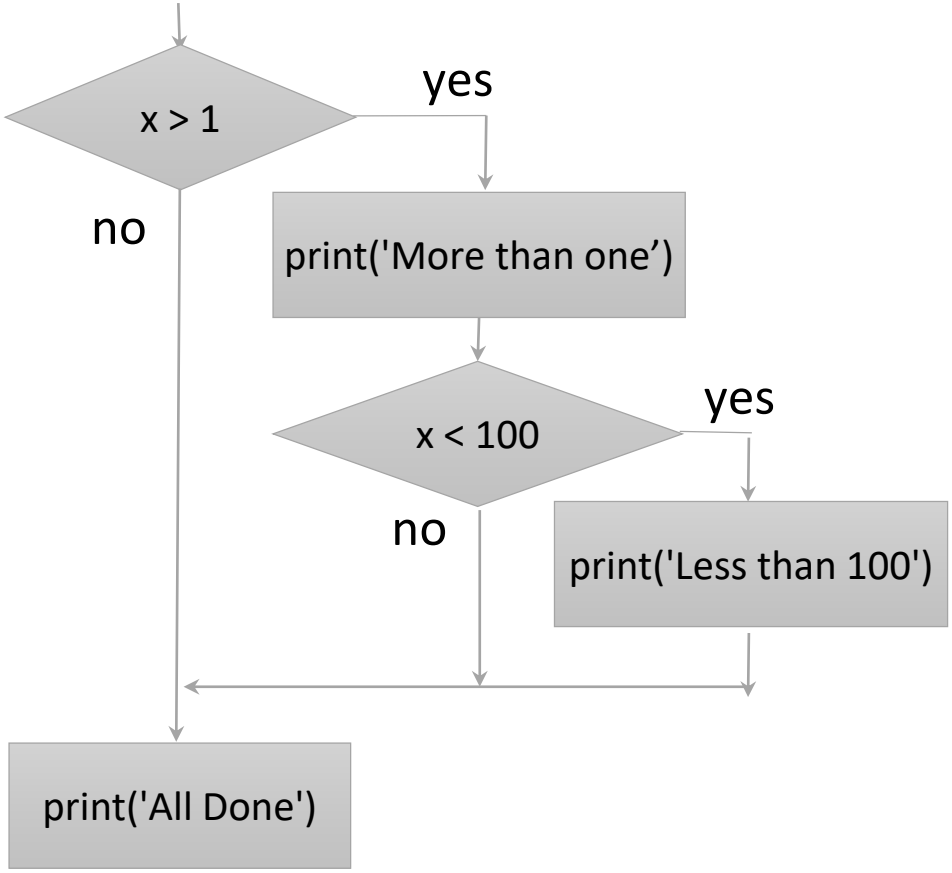
Indentation

- Maintain indent to indicate the scope of the block (which lines are affected by the **`if`**).
- Reduce indent back to the level of the **`if`** statement to indicate the end of the block.

```
x = 5
if x > 2:
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5):
    print(i)
    if i > 2:
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```

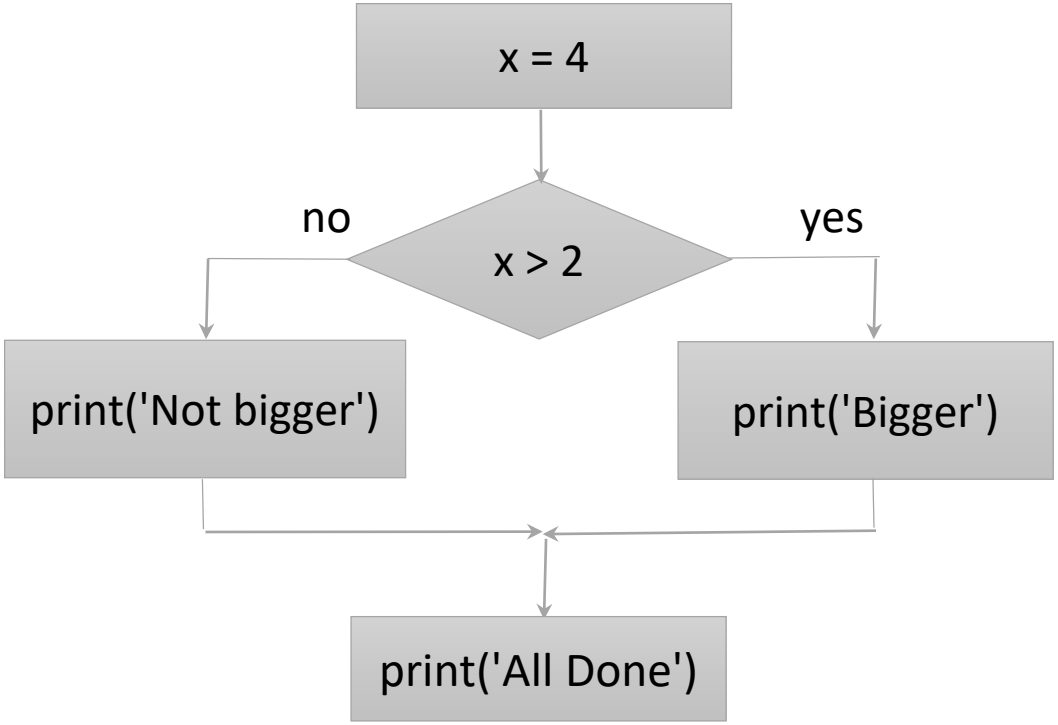
Nested Conditionals

```
x = 42
if x > 1:
    print('More than one')
    if x < 100:
        print('Less than 100')
print('All Done')
```



Chained Conditionals

- Sometimes we want to do one thing if a Boolean expression is true and something else if the expression is false.
- It is like a fork in the road - we must choose one or the other path but not both.

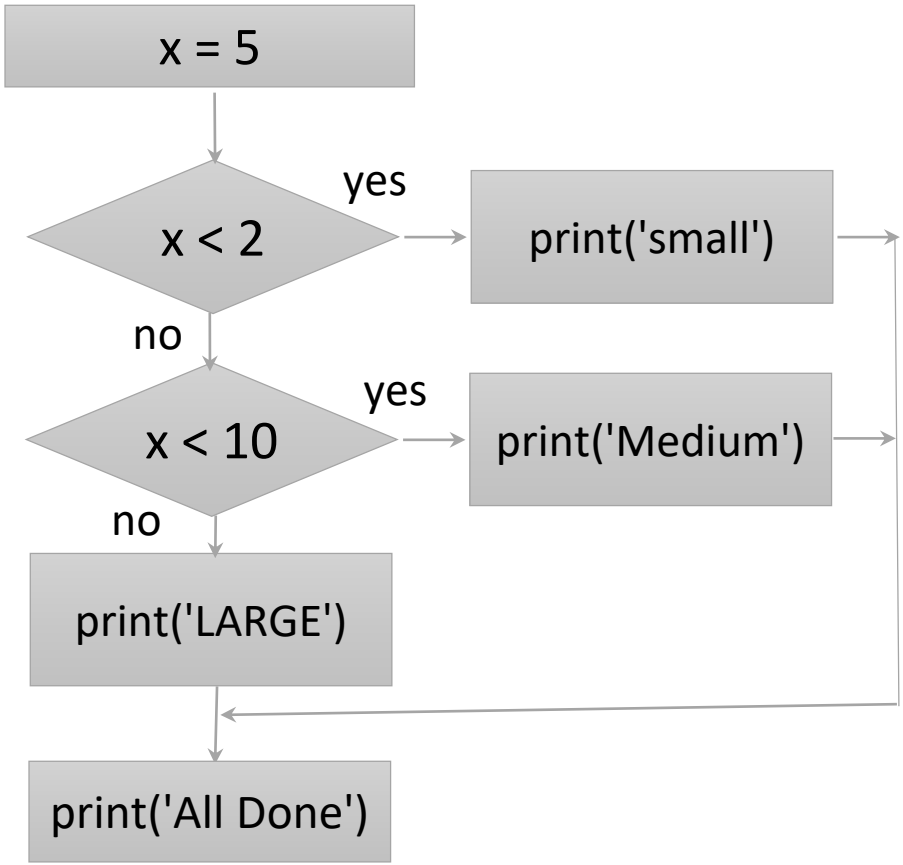



```
x = 4

if x > 2:
    print('Bigger')
else:
    print('Smaller')
print('All Done')
```

Chained Conditionals

- When having more than two options:

```
x = 5
if x < 2:
    print('small')
elif x < 10:
    print('Medium')
else:
    print('LARGE')
print('All done')
```



Functions

Python Functions

- There are two kinds of functions in Python.
  - Built-in functions that are provided as part of Python - `print`, `input`, `type`, `float`, `int` …
  - Functions that we define ourselves and then use
- We treat the built-in function names as "new" reserved words (i.e., we avoid them as variable names).
- There are also built-in modules, which are a collection of related functions.

Common Built-in Functions

Functions	Description
abs()	Returns the absolute value of the given number and returns a magnitude of a complex number.
sorted()	Returns a sorted list from the items in an iterable.
round()	Returns a floating-point number rounded to the specified number of decimal point.
max()	Returns the biggest item.
len()	Returns the number of items in a container.
sum()	Returns the total of integer elements starting from left to right of the given iterable.
bool(x)	Returns True when the argument x is true, False otherwise.
all(x)	Returns True if bool(x) is True for all values x in the iterable.
any(x)	Returns True if bool(x) is True for any x in the iterable.
bin()	Converts an integer number to a binary string prefixed with '0b'.
id()	Returns the unique identity of an object.
range()	Returns a sequence of integers.

Build Our Own Functions

- We create a new function using the **`def`** keyword followed by optional parameters in parentheses.
- This defines the function but does not execute the body of the function.
- Once we have defined a function, we can call (or invoke) it as many times as we like.

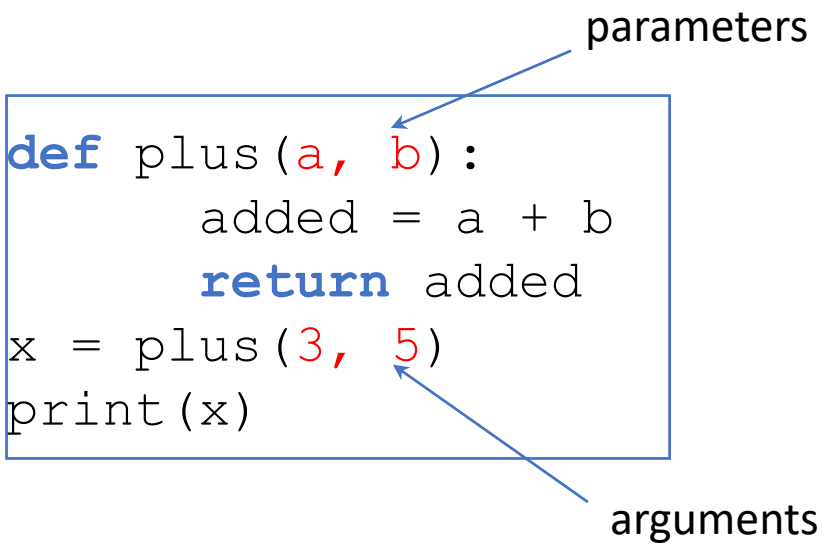
```
x = 5
print('start')

def msg(name):
    print("hello", name)

msg("Chao")
x = x + 2
print(x)
```

```
start
hello Chao
7
```

Arguments & Parameters

- An argument is a value we pass into the function as its input when we call the function.

- A parameter is a variable which we use in the function definition. It is a "handle" that allows the code in the function to access the arguments for a particular function invocation.



Return Values

- Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The **`return`** keyword is used for this purpose.
- A "fruitful" function is one that produces a result.
- When a function does not return a value, we call it a "void" function, also called a non-fruitful function.

```
def greet():
    return "Hello"

print(greet(), "Mr. Chan")
print(greet(), "Miss Lee")
```

Hello Mr. Chan
Hello Miss Lee

Lambda Expressions

- Lambda expressions are small and anonymous functions.
- Only one expression in the lambda body; its value is always returned.
- Lambda expressions can be used in many other functions such as **`filter`** and **`map`**.

```
addone = lambda x: x + 1
addone(100)
```

101

```
sumup = lambda x, y: x + y
sumup(10, -6)
```

4

```
evenodd = (lambda x: x % 2)(51)
print(evenodd)
```

1

To Function or not to Function...

- Organize your code into “paragraphs” - capture a complete thought and “name it”.
- Don’t repeat yourself - make it work once and then reuse it.
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions.
- Make a library of common stuff that you do over and over - perhaps share this with your friends...

Module and Packages

Modules

- A module is a collection of related functions (also called methods).
- It can be imported and used elsewhere.

```
import math
from math import pi, sin
from random import random as rd
from random import *
```

- When calling the functions, the names will be different depending on how they are imported.

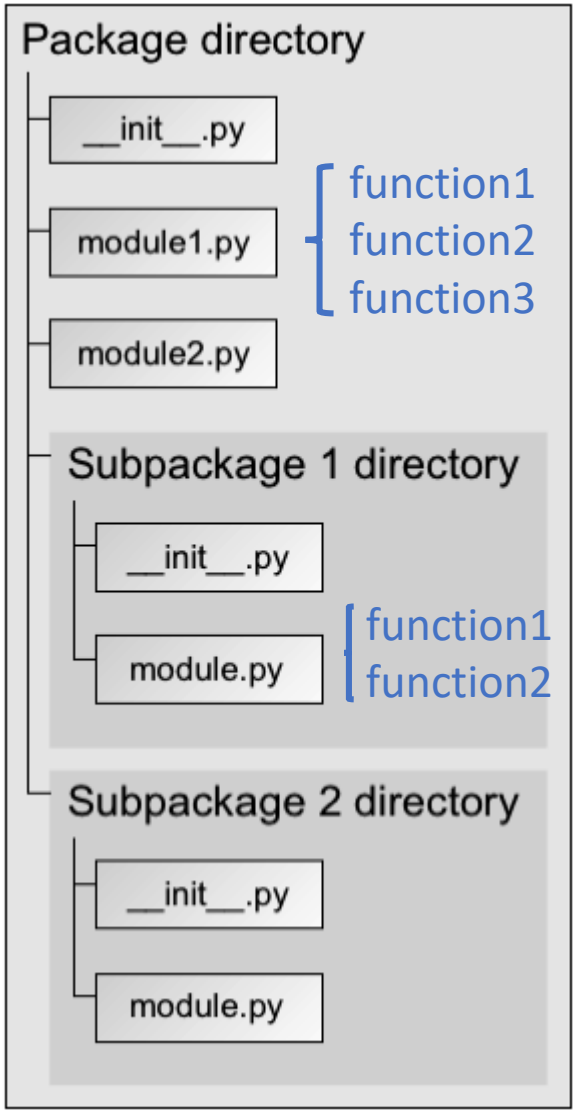
Packages

- We can import a module from a package using the dot notation.

```
import package.module
from package.module import function
```

- Packages themselves may contain packages; these are subpackages. To access subpackages, you just need to use a few more dots.

```
import package.subpackage.module
from package.subpackage import module
```



`math` Module

- The `math` module provides mathematical functions.

```
>>> import math

>>> math.sin(45)
0.7071067811865476

>>> math.sqrt(2) / 2.0
0.7071067811865476

>>> math.floor(5.6)
5

>>> math.ceil(5.6)
6

>>> math.log10(100)
2.0
```

Function	Description
ceil(x)	Rounds x up to the nearest integer
floor(x)	Rounds x down to the nearest integer
cos(x)	Returns the cosine of a number
sin(x)	Returns the sine of a number
tan(x)	Returns the tangent of a number
degrees(x)	Converts an angle from radians to degrees
radians(x)	Converts a degree value into radians
exp(x)	Returns E raised to the power of x
sqrt(x)	Returns the square root of x
log(x)	Returns the natural logarithm of x
log10()	Returns the base-10 logarithm of x

`random` Module

- The `random` module provides functions that generate pseudorandom numbers .

Function	Description
seed()	Initializes the random number generator
getrandbits()	Returns a number representing the random bits
randrange()	Returns a random number between the given range
randint()	Returns a random number between the given range
choice()	Returns a random element from the given sequence
choices()	Returns a list with a random selection from the given sequence
shuffle()	Takes a sequence and returns the sequence in a random order
sample()	Returns a given sample of a sequence
random()	Returns a random float number between 0 and 1
uniform()	Returns a random float number between two given parameters

`string` Module

- The `string` module contains functions which produce string constants.
- Note, there is no parenthesis when retrieving a constant.

Function	Description
ascii_uppercase	The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
ascii_lowercase	The lowercase letters 'abcdefghijklmnopqrstuvwxyz'.
ascii_letters	The combination of lowercase and uppercase.
digits	The string '0123456789'.
hexdigits	The string '0123456789abcdefABCDEF'.
punctuation	String of ASCII characters which are considered punctuation characters in the C locale: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.
whitespace	A string containing all ASCII characters that are considered whitespace such as space, tab, linefeed, return, and vertical tab.
printable	The combination of digits, ascii_letters, punctuation, and whitespace.

Other Useful and Simple Built-in Modules

- `collections`: provides different types of containers.
- `time`: provides functions related to time.
- `datetime`: provides functions to work with date and time.
- `calendar`: provides functions related to calendar.
- `statistics`: provides functions to mathematical statistics of numeric data.

Loops and Iterations

Loops and Iterations

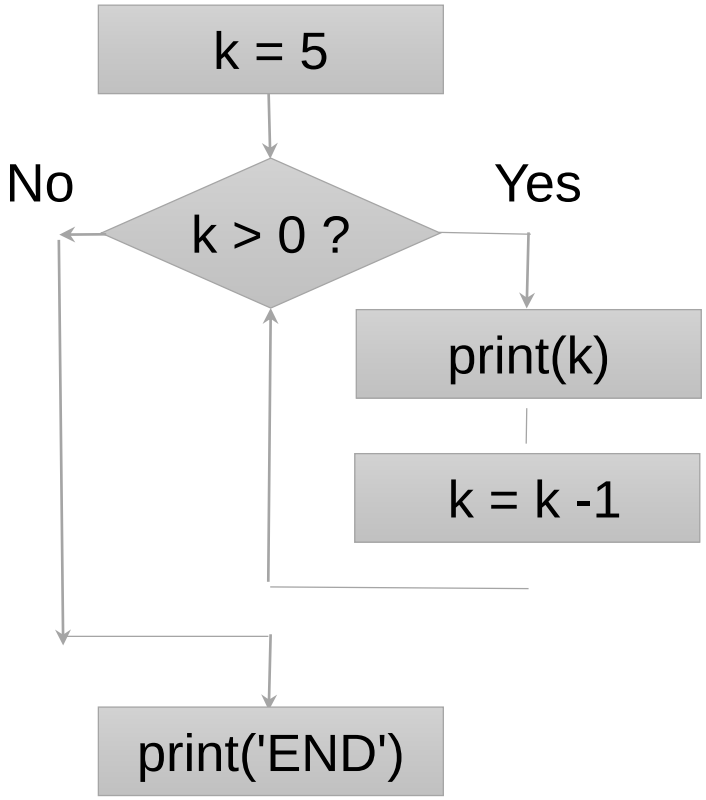
- Introduce nonlinearity into programs.
- Repeatedly execute blocks of code.
- Two general categories:
  - Condition-controlled loops, the `while` loops.
  - Count-controlled loops, the `for` loops.

The `while` Loop

- Executes a set of statements as long as a condition is True.
- Loops have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.

```
k = 5
while k > 0:
    print(k)
    k = k - 1
print('END')
print(k)
```

```
5
4
3
2
1
END
0
```



Break Out of a Loop

- The **`break`** statement ends the current loop and jumps to the statement immediately following the loop.
- It is like a loop test that can happen anywhere in the body of the loop.

```
import random

while True:
    k = random.randint(1, 5)
    print(k)
    if k == 4: break
print('The End!')
```

```
1
5
2
1
1
3
4
The End!
```

Stop an Iteration with `continue`

- The **`continue`** statement ends the current iteration and jumps to the top of the loop and starts the next iteration.

```
import random

while True:
    gpa = round(random.random() * 4, 1)
    print(gpa)
    if gpa < 3: continue
    else: break

print('The End!')
```

1.0
2.5
3.9
The End!

The Most Common Rookie Mistakes

```
n = 5
while n > 0:
    print('Lather')
    print('Rinse')
print('Dry off!')
```

Infinite loop

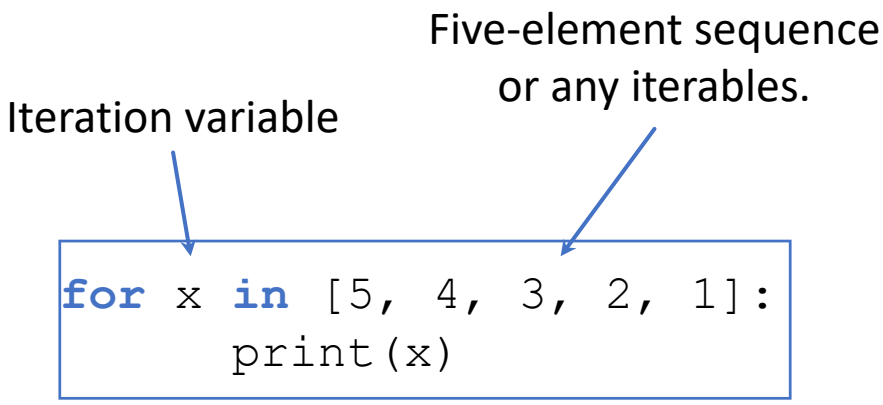
```
n = 0
while n > 0:
    print('Lather')
    print('Rinse')
print('Dry off!')
```

Never starts

- Tips:
  - Print some message to know that it's working.
  - Make use of break and continue.
  - Use a small sample to test first.

The `for` Loop

- The iteration variable "iterates" through the sequence of values (called an iterable) one by one.
- The body is executed once for each value in the sequence.




```
5
4
3
2
1
```

Nested `for` Loops

- Body of the `for` loop contains another `for` loop.
- Second loop must be completely contained inside the first loop. Second loop must have a different iteration variable.

```
for x in range(3):
    for y in range(4):
        print(y)
    print()
```

```
0
1
2
3

0
1
2
3

0
1
2
3
```

Outer loop determines rows

Inner loop determines columns

```
for x in range(3):
    for y in range(4):
        print(y, end=' ')
    print()
```

```
0 1 2 3
0 1 2 3
0 1 2 3
```

Choose Between `for` and `while`

- `while` loops are called "indefinite loops" because they keep going until a logical condition becomes False.
- Use a `while` loop if you are required to repeat some computation until some condition is met.
  - repeat this task as long as the light is on
  - keep searching until it is found
- `for` loops are called "definite loops" because they execute an exact number of times.
- Use a `for` loop if you know, before you start looping, the maximum number of times that you'll need to execute the body.
  - iterate this calculation for 1000 times
  - search this list of 200 words

Strings

Define a String

- A string is a sequence of characters and is defined by quotes.

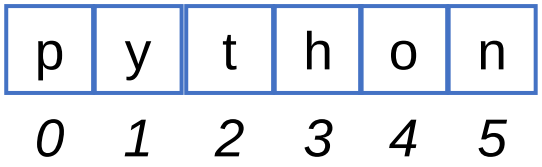
```
str1 = 'hello'
str2 = "hello"
str3 = """hello"""
```

- We can also use **`str`** to create a string.

```
str4 = str()
str5 = str(123)
```

String Indexing

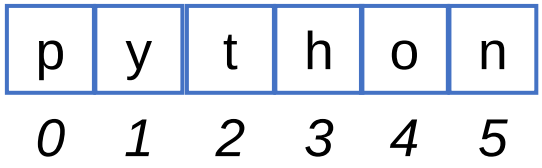
- Every character in a string has an integer index, starting at zero. We use square brackets [index] to find the element.
- The last character is also indexed -1, second to the last is also indexed -2, and so on.
- You get an IndexError if you go beyond the end of a string. The built-in function **`len`** gives us the length of a string.




```
>>> s = ' python'
>>> print(s[1])
y
>>> print(s[-2])
o
>>> print(len(s))
6
>>> print(s[10])
IndexError: string index out of range
```

String Slicing

- To look at any slice, use colon operator: [start:stop:stepSize]
- stop means “up to but not including”.
- If stop is beyond the end of the string, it stops at the end.
- If we leave off the start or the stop, it is assumed to be the beginning or end of the string respectively.
- stepSize means the number of spaces between successive characters. i.e., 2 means every other character.




```
>>> print(s[0:4])
pyth
>>> print(s[4:5])
o
>>> print(s[4:30])
on

>>> print(s[:2])
py
>>> print(s[3:])
hon
>>> print(s[:-1])
pytho

>>> print(s[::2])
pto
```

Strings are Immutable

- Strings are immutable. Characters cannot be revised.

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not
support item assignment
```

- The + operator means “concatenation”.
- The * operator means “replication”.

```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print(b)
HelloThere

>>> c = a * 3
>>> print(c)
HelloHelloHello
```

Traverse a String

- Strings are considered iterables.
- Using a **`for`** or **`while`** loop, an iteration variable, and the **`len`** function, we can construct a loop to look at each of the characters in a string individually.

```
fruit = 'banana'
for letter in fruit:
    print(letter, end = ' ')
```

```
b a n a n a
```

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(index, letter)
    index = index + 1
```

```
0 b
1 a
2 n
3 a
4 n
5 a
```

String Methods

- Python has a number of string methods (functions).
- These methods are already built into every string object.
- We invoke them by appending the method to the string variable using dot as a delimiter.
- String methods do not modify the original string, instead they return a new string that has been altered.

```
>>> greet = 'Hello Bob'
>>> print(greet.lower())
hello bob
>>> print(greet)
Hello Bob
>>> print('Hi There'.upper())
HI THERE
```

Common String Methods

Method	Description
capitalize()	Converts the first character to upper case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
endswith()	Returns True if the string ends with the specified value
find()	Searches the string for a specified value and returns the position of where it was found
join()	Converts the elements of an iterable into a string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
startswith()	Returns True if the string starts with the specified value
strip()	Returns a trimmed version of the string
upper()	Converts a string into upper case

More String Methods

- Methods to evaluate the characters in a string.

Method	Description
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
isupper()	Returns True if all characters in the string are upper case

f-strings

- f-strings are a concise way to evaluate formatted string expressions. They start with a preface of f (or F).
- Variables & expressions go into {} to be evaluated.

```
>>> name, gpa = "Chao", 4.1
>>> f"Hello, {name}. Your GPA is {gpa}."
'Hello, Chao. Your GPA is 4.1.'
```

- Within {}, one may further specify the alignment, width, and representation (e.g., float, percentage, binary).

{variable:alignment width representation}

```
>>> print(f"Hello, {name:^10}. Your GPA is {gpa:.2f}")
Hello,   Chao    . Your GPA is 4.10.
```

```
>>> print(F"Hello, {name.upper()}. Your GPA is {gpa:>15%}.")
Hello, CHAO. Your GPA is     410.000000%.
```

Lists

Define a List

- Like a string, a list is a sequence of values, called elements.
- A list element can be any type - even another list.
- We use brackets [ ] or the **`list`** function to create a list.

```
majors = ['A&F', 'ECON', 'BA']
numbers = [1, [5, 6], 7]
names = list()
```

- Just like strings, we could do list indexing or list slicing where the second number is "up to but not including".

```
>>> print(majors[2])
BA
>>> print(majors[1:])
['ECON', 'BA']
```

Lists are Mutable

- Lists are mutable. Elements can be revised.

```
>>> lotto = [2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print(lotto)
[2, 14, 28, 41, 63]
```

- The + operator means “concatenation”.
- The * operator means “replication”.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a * 2)
[1, 2, 3, 1, 2, 3]
```

Traverse a List

- Use a **`for`** loop to either look at every element, or look at every index.

```
majors = ['A&F', 'ECON', 'BA']
for major in majors:
    print(major)
```

```
for i in range(len(majors)):
    major = majors[i]
    print(major)
```

A&F

ECON

BA

List Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Adds the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tuples

Define a Tuple

- Tuples are another kind of sequence that functions much like a list - index starting at 0.
- To create a tuple, we just need to create different comma-separated values. Or, put them in parentheses.

```
>>> tup1 = ('english', 'french', 'spanish', 1997, 2000)
>>> tup2 = (1,2,3,4,5)
>>> tup3 = 'a', 'b', 'c', 'd'
>>> tup4 = ()
>>> tup5 = tuple()
```

- To create a tuple containing a single element, you have to include a comma.

```
>>> tup5=(50, )
```

Tuples are Immutable

- Similar to a string, tuple elements can't be revised.

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple' object does not support item Assignment
```

- Tuples have limited methods.

```
>>> dir(list)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> dir(tuple)
['count', 'index']
```

- We can convert between a list and a tuple using the built-in **`tuple`** or **`list`** function.

Sets

Define a Set

- A set is a collection of values that are unordered and unindexed.
- We can create a set using curly braces {}, or **`set`** function.
- Duplicates are not allowed in a set.

```
>>> nums = {4, 6, 9, 4}
>>> print(nums)
{9, 4, 6}
```

- 1 and **`True`**, 0 and **`False`** are considered the same value.

```
>>> values = set(['apple', 1, False, 'tree', 0, -19, True])
>>> print(values)
{False, 1, 'tree', -19, 'apple'}
```

Sets are Somewhat Immutable

- Set elements do not have index.

```
>>> nums[0]
TypeError: 'set' object is not subscriptable
```

- Set elements are unchangeable, but you can remove elements and add new elements with set methods.

```
>>> nums.add(-20)
>>> print(nums)
{9, 4, -20, 6}
```

```
>>> nums.discard(9)
>>> print(nums)
{4, -20, 6}
```

- The * and + operators do not work on sets. Use union operator | or the .union method to merge multiple sets.

```
>>> nums * 2
>>> nums + {-100, 99}
```

```
>>> nums | values
{9, 4, 6, False, 1, 'tree', -19,
'apple'}}
```

Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Removes the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
union()	Returns a set containing the union of sets
update()	Updates the set with the union of this set and others

Dictionaries

Define a Dictionary

- A dictionary is a mapping between indices (which are called keys) and values. Each key maps to a value.
- Dictionary items are key: value pairs. Keys must be unique.
- We define a dictionary by curly braces {} or **`dict`** function.

```
>>> empDict = {} # or, dict()
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

- We look up a value by "indexing" its key. It is an error to reference a key not in the dictionary.

```
>>> eng2sp['one']
'uno'
```

```
>>> eng2sp['five']
KeyError: 'Five' does not exist
```

Key-value Pairs

- To add and update key-value pairs.

```
>>> grades = dict()
>>> grades['Chao'] = 97
>>> grades['Eric'] = 82
>>> print(grades)
{'Chao': 97, 'Eric': 82}

>>> grades['Chao'] = 95
>>> print(grades)
{'Chao': 95, 'Eric': 82}
```

- To merge multiple dictionaries, use the union operator | or the ** operator.

```
x = {'Wong': 28, 'Lee': 50}
y = {'Chan': 34}
```

```
x | y
```

```
{**x, **y}
```

{'Wong': 28, 'Lee': 50, 'Chan': 34}

Traverse Dictionary Keys

- We use the **`in`** operator to see if a key is in the dictionary.

```
>>> 'Chuck' in grades
False
```

- We use a **`for`** loop to go through all of the keys in the dictionary and look up the values.

```
for name in grades:
    print(f'{name} earned {grades[name]} points.')
```

Chao earned 95 points.
Eric earned 82 points.

Complex Dictionaries

- The values of a dictionary can also be lists or dictionaries.

```
departments = {
'business': ['accounting','finance','economics'],
'linguistics': ['chinese','english','japanese']
}
departments['linguistics'][2]
```

'japanese'

```
people = {
1: {'Name': 'John', 'Age': '27', 'Sex': 'Male'},
2: {'Name': 'Marie', 'Age': '22', 'Sex': 'Female'}
}
for i in people.values():
    print(i['Name'])
```

John
Marie

Dictionary Methods

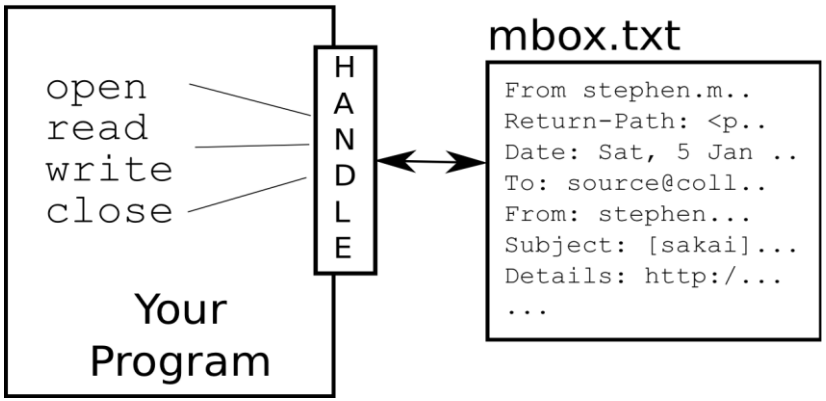
Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a sequence containing a tuple for each key value pair
keys()	Returns a sequence containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a sequence of all the values in the dictionary

File Input/Output (I/O)

Opening a File

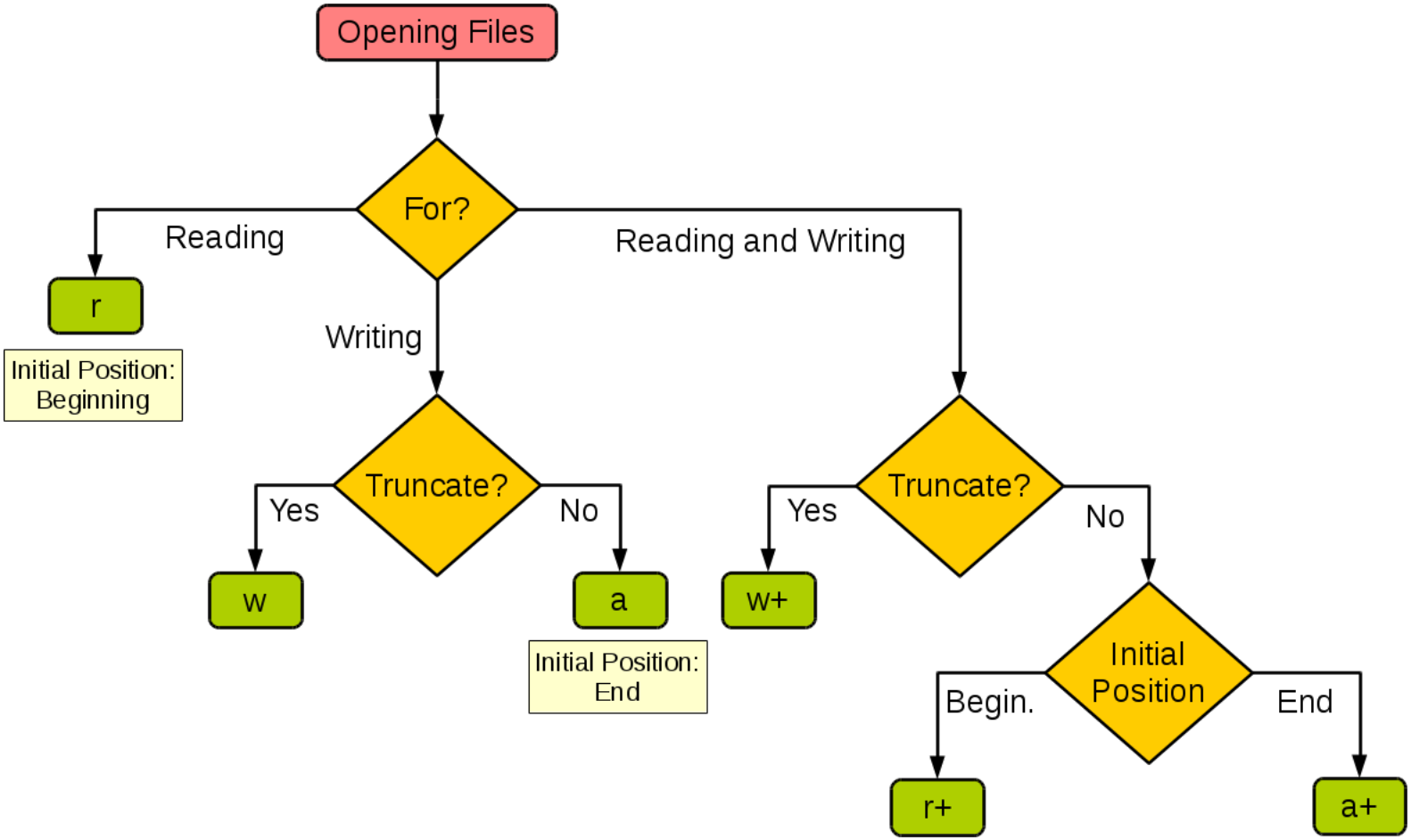
- Before we can read the contents of the file (txt or csv or any Python supported files), we must tell Python which file we are going to work with.
- `open` returns a "file handle" (an io.TextIOWrapper object) - a variable used to perform operations on the file.

```
open(filepath/filename, mode)
```



About Mode

- Default mode is **r**. There is also a binary mode **b**, which forces the data to be binary.



Filenames and Paths

- If the file is in the same directory with your code, simply type the file name.

```
handle = open('file.txt', 'r')
```

- If it's in the parallel directory, then use relative path.

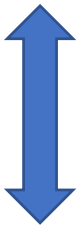
```
handle = open('../data_in/file.txt', 'r')
```

- Or, you can also use the absolute path, by using the full path of the file such as "c:/home/data/file.txt".

Closing a File

- After working with the file, we should always close it with the **`close`** method.
- Or, a shortcut by using the **`with`** statement. It will take care of closing the file.

```
>>> handle = open('file.txt','r')
>>> do something with the file
>>> handle.close()
```



Equivalent

```
with open('file.txt','r') as handle:
    do something with the file
```

File Methods

Method	Description
close()	Closes the file
readable()	Returns whether the file stream can be read or not
seek()	Changes the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not

TXT

Text File Structure

- A file handle opens for read can be treated as a sequence of strings where each line in the file is a string in the sequence.
- There is a newline at the end of each line.

```
Two roads diverged in a yellow wood,\n
And sorry I could not travel both\n
And be one traveler, long I stood\n
And looked down one as far as I could\n
To where it bent in the undergrowth; \n
```

This is an excerpt from io_txt.txt

- We can use the `for` loop to traverse the sequence.

```
handle = open('file.txt', 'r')
for line in handle:
    print(line)
```

Methods to Read and Write

Method	Description
read()	Returns the file content as one string
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

CSV

CSV File Structure

- CSV: **C**omma **S**eparated **V**alues
- It is a readable text file.
- Every row is considered as a list or a tuple.
- So, the file content is a list of lists (tuples).
- The first row is usually a header row.

Opened in text editor

```
no, Name, City
1, Michael, New Jersey
2, Jack, California
3, Donald, Texas
```

Opened in Excel

no	Name	City
1	Michael	New Jersey
2	Jack	California
3	Donald	Texas

Methods to Read and Write

- We need to use the **`csv`** module.
- First create a reader/writer object and then read/write via the reader/writer object.

```
import csv

reader = csv.reader(handle)
writer = csv.writer(handle)
```

These are methods of the writer object

Method	Description
writerow()	Writes the specified list (tuple) to the file
writerows()	Writes a list of lists (tuples) to the file

- There are also methods to read and write dictionaries: **`csv.DictReader`** and **`csv.DictWriter`**.

JSON

JSON File Structure

- JSON: **J**ava**S**cript **O**bject **N**otation.
- JSON is a syntax for storing and exchanging data.
- It is a very popular way of exchanging data online.
- It is still considered text, also called JSON strings.

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

```
[
 {"sub1":"math", "status":"successful"},
 {"sub2":"stat", "status":"failed"}
]
```

JSON Syntax Rules

- Data is in key/value pairs.
- keys must be strings, written with double quotes.
- values must be one of the following data types:
  - a string `{ "name":"John" }`
  - a number `{ "age":30 }`
  - an object another JSON object
  - an array `{"employees":[ "John", "Anna", "Peter"]}`
  - a boolean `{ "sale":true }`
  - null `{ "middlename":null }`

JSON Strings vs. Python Objects

- There is a one-on-one conversion between JSON strings and Python objects.

JSON String	Python Object
object	dict or list
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Methods to Read and Write

```
import json
```

Method	Description
load()	Returns the file content as a Python dictionary.
loads()	Converts a JSON string into a Python object.
dump()	Writes a Python dictionary to the file.
dumps()	Converts a Python dictionary into a JSON string.

One Liners

```python
# swap two variables
a, b = b, a

# list creation with a for loop
MyList = [num ** 3 for num in range(9)]

# conditionally assign values to a variable
PorF = 'P' if grade > 60 else 'F'

# condition and for loop in one line
PassFail = [grade > 60 for grade in all_grades]
HowManyPass = sum(grade > 60 for grade in all_grades)
IndexPass = [index for index, grade in enumerate(all_grades) if grade > 60]
WhoPass = [name for name, grade in zip(all_names, all_grades) if grade > 60]

# check if all elements or any element meet a condition
all(num % 2 == 0 for num in all_nums)
any(num in target_nums for num in all_nums)

# concatenate string elements
'+'.join(list('Hello World'))
''.join(handle.readlines())
'\n'.join(line for line in all_lines)
```

See more at: https://www.logicalpython.com/powerful-python-one-liners-for-daily-use/

Object Oriented Programming (OOP)

What is OOP?

- Python is an object oriented programming (OOP) language.
- Almost everything in Python is an object or an instance of a certain class (type), with its attributes and methods.
- attributes refer to the properties or data associated with a specific object of a given class.
- methods refer to the different behaviors that objects will show.

Class	Instance	Instance attribute	Instance method
str	name = "Chao" gender = "Male"		name.upper() gender.replace('M', 'm')
io.TextIOWrapper	handle1 = open(path1) Handle2 = open(path2)	handle1.name handle2.mode	handle1.seek(0) handle2.read(5)

What is a Class?

- You can think of a class as a piece of code that specifies the data and behavior that represent and model a particular type of object.
- A class is like an object constructor, or a "blueprint" for creating objects.
- Built-in classes we already learned:

```
int, float, str, list, tuple, set,
dict, range, bool, zip, enumerate,
map, filter, io.TextIOWrapper …
```

Create a Class

- We use the **`class`** keyword to define a class. Then, create instances, known as instantiation.

```
class student:
    course = "MSBA7001"

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def __str__(self):
        return f"{self.name} earned {self.score} in {self.course}."

    def passed(self, cutoff = 75):
        return True if self.score > cutoff else False

    def printout(self):
        if self.passed():
            print(f"{self.name} passed the course.")
        else:
            print(f"{self.name} failed the course.")
```

Attributes and String Representation

- Class attributes are shared among all instance.
- Instance attributes are attached to the instance itself.
- String representation always returns a user-friendly string.
- We use the dunder (double underscore) method to define instance attributes and string representation.
- self here means the instance itself.

```
class student:
    course = "MSBA7001"

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def __str__(self):
        return f"{self.name} earned {self.score} in {self.course}."
```

Class attribute

Instance attribute

String representation

Attributes and String Representation

```
class student:
    course = "MSBA7001"

    def __init__(self, value1, value2):
        self.name = value1
        self.score = value2

    def __str__(self):
        return f"{self.name} earned {self.score} in {self.course}."
```

Instantiation

```
chao = student("Chao", 90)
eric = student("Eric", 60)
```

Class attribute

```
student.course, chao.course
```

('MSBA7001', 'MSBA7001')

Instance attribute

```
chao.name, eric.score
```

('Chao', 60)

String representation

```
print(chao)
```

Chao earned 90 in MSBA7001.

Methods

- To define an instance method, you just need to write a regular function that accepts self as its first argument.
- Instance methods should act on instance attributes by either accessing them or changing their values.
- Note there are also class methods and static methods.

```
class student:
    # …

    def passed(self, cutoff = 75):
        return True if self.score > cutoff else False

    def printout(self):
        if self.passed():
            print(f"{self.name} passed the course.")
        else:
            print(f"{self.name} failed the course.")
```

Methods

```
class student:
    # …

    def passed(self, cutoff = 75):
        return True if self.score > cutoff else False

    def printout(self):
        if self.passed():
            print(f"{self.name} passed the course.")
        else:
            print(f"{self.name} failed the course.")
```

`chao.passed()`	True
`eric.passed()`	False
`chao.printout()`	Chao passed the course.
`eric.printout()`	Eric failed the course.

Import a Class

- User-defined classes can be saved as a python file (.py).
- We can later import the class just like importing a module.
- This is exactly how we may share self-built modules and classes with others.


```
from example import student

meili = student("Meili", 88)
meili.printout()
```

- OOP is a relatively advanced topic. To learn more, you are advised to study this article: Python Classes: The Power of Object-Oriented Programming

Get Ready for MSBA7001

1. Install Anaconda (Python)
   - https://www.anaconda.com/download

2. Install VS Code (IDE)
   - https://code.visualstudio.com/download

3. Register a GitHub Account
   - https://github.com/

4. Join GitHub Education (GitHub Copilot)
   - https://github.com/education/students

THE END.