Python

The canonical, "Python is a great first language", elicited, "Python is a great last language!"

- Noah Spurrier

What is Programming Language?

The language of computers

How to make computers do the computation we want

Levels:

- Machine level
- High level
- Natural Language

Where does Python fit in?

High level, general purpose

Interpreted

Dynamically typed

Multiple paradigms: object oriented, imperative, functional, procedural

Interpreter vs Compiler

Everything is just bytes

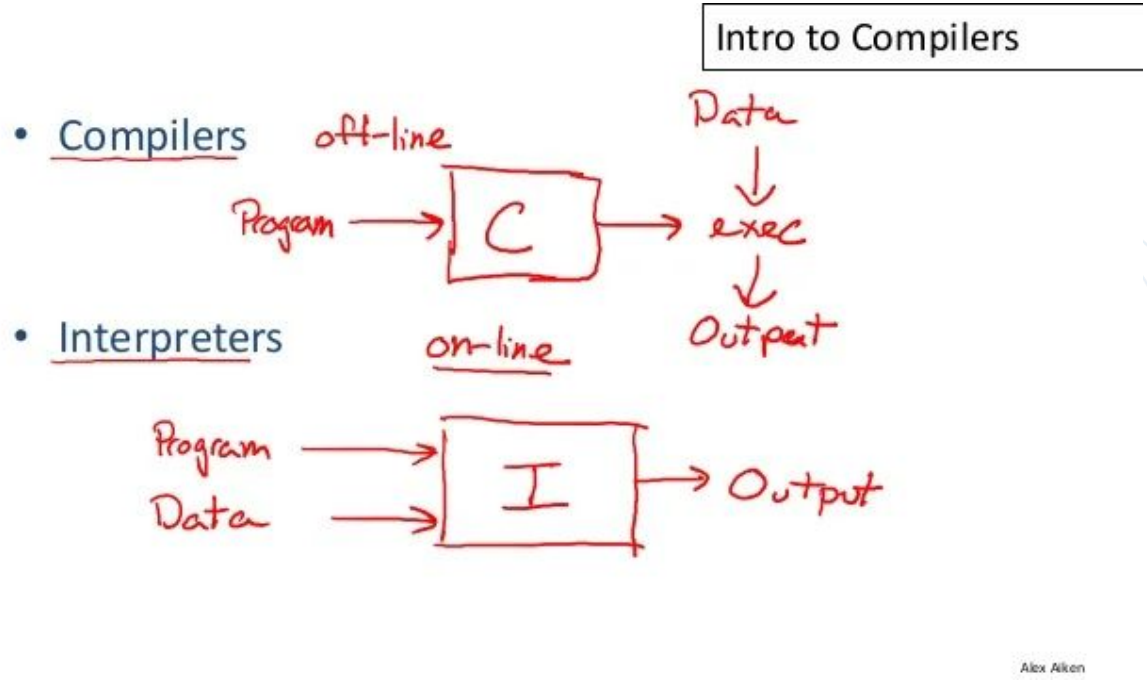
The C compiler gcc is a program

The Python interpreter is a program

A program is just bytes in memory

xxd /usr/bin/python

Interpreter vs Compiler



Source: Alex Aiken's course (Compilers)

Interpreter vs Compiler

Python is slow
The Interpreter itself ships with every program!

Compiled programs are faster (Can be optimized to low level machine code that does exactly what is needed)

So why python?

Python is fast enough!
Python can call other programs if needed!
Python provides high level features for us programmers!
Python is a beautiful language
Python is easy(sort of)

Python Shell Invocation

How to invoke a program?

Just type the program's name (It must be in the PATH)

How to invoke python?

Just type python (After you install it and add it to PATH)

Run a python script

Python files end in .py by default

Python will execute each instruction line by line

How do I run a python file?

python <file name>

Python REPL

REPL(Read Eval Print Loop)

- Useful for testing features
- Has some additional features
- Sort of a playground
- Is a handy calculator :)

Python program options

python3 - - version

python3 -c “print(‘apple’)”

from __future__ import braces

Python will never have braces!

Not a chance

Python uses whitespace for scoping (spaces not tabs a/c to PEP-8)

Variables & Data types

Variables are names given to buckets

Buckets can hold different things

The things they hold can change

- Variables → names of buckets
- Data types → buckets

Static typing: a bucket can hold only one type of thing

Dynamic typing: a bucket can hold any thing

Variables & Data types

Defining a variable

```
a = 4
Speech = “Python is awesome!”
_floating_variable_4 = 4.0
```

Names are case sensitive!

Names can have numbers(not at the beginning) and underscore(_)

Everything you didn’t define yourself is a builtin!

The type builtin shows which buckets a variable belongs to!

Variables & Data types

Python is dynamic

A = 4 (A is an integer type)

A = “Python” (A is now the string type!)

The difference between 3 and “3”

3 is an integer

“3” is a string

Variables & Data types

Literals in Python

the raw data assigned to variables

Five types of literals

string literals “python”

numeric literals 4.6, 4+5j

boolean literals True, False

literal collections: Lists [1,3,4], tuples, dictionaries, sets

a special literal: None

Variables & Data types

Casting

Convert 3 to “3”

str(3)

Convert “3” to 3

int(“3”)

For floating point numbers → float(“4.5”)

User Input and Output

```
username = input("Enter your name")
print(“Hi “+ username)
```

Complete syntax

```
print(object(s), sep=separator, end=end)
```

Separator : what to show in between 2 objects

End : what to show at the end

Basic operators and operator precedence

Include

- to perform operations on variables and values
- Arithmetic operators: +,-,*,/,%,**,//
- Assignment operators: =,+=,-=, and so on
- Comparison operators: >, <,<=,>=,==,!=
- Logical operators : and , or , not
- ternary operator: a if b else c
- Others: membership operators,bitwise operators,Identity operators

Operator Precedence

- Which operation to carry first when more than one operators are involved in an expression?

```
10 + 20 * 30 # calculated as 10 + (20 * 30)
```

- Order of Precedence

  Parenthesis (), Exponential **, Multiplication *, Division /, Addition +, Subtraction -

  https://www.tutorialspoint.com/python/operators_precedence_example.htm

String Operations

String operations(startswith, split, indexing, slicing, formatting etc.)

Indexing : “Python string”[3]

Slicing :“Python string”[start:stop:step]

“Python string”.startswith(“Pyth”)

“Python string”.endswith(“ing”)

Formatting

```
'Hello, {}'.format(name)
f'Hello, {name}!'
```

Flow Control

Order in which the program's code executes

- Sequential - default
- Selection - decision and branching
- Repetition - looping

if-elif

```
x = 15
y = 12
if x == y:
   print("Both are Equal")
elif x > y:
    print("x is greater than y")
else:
    print("x is smaller than y")
```

Looping

- to repeat a group(block) of programming instructions
- For loop(iterate over sequence)
- While loop(repeat until the given condition is satisfied)

```
lst = [1, 2, 3, 4, 5]
for i in range(len(lst)):
     print(lst[i], end = " ")

lst = [1, 2, 3, 4, 5]
for i in lst:
     print(i, end = " ")
```

```
m = 5
i = 0
while i < m:
     print(i, end = " ")
     i = i + 1
print("End")
```

Looping

Break out of the loop with `break`

Skip the remaining part of the loop with `continue`

```
for i in range(10):
    print("before")
    print(i)
    continue
    print("after")
```

after will never be printed

```
for i in range(10):
    print("before")
    print(i)
    break
    print("after")
```

Only print one time

Truthy and Falsy Values

`None`

`False`

0, or any numerical value equivalent to zero, for example 0, 0.0, 0j

Empty sequences: `''`, `""`, (), []

Empty mappings: `{}`

User-defined types where the `__bool__` or `__len__` methods return 0 or False

The above list shows all falsy values all other values are truthy!

Lists, Tuples, Dictionaries and Sets

More data types

Python has Lists, Dictionary, Tuples and Set builtin!

[List] is a list of things (obviously)

{Dictionary} is a mapping to from a hashable thing to another thing

(Tuples) are immutable lists

{Set} is like the set from set theory (math)

List and Tuple operations

Slicing: Same as string slicing

```
mylist[start:stop:step]
```

Trick: Reverse a list with `[::-1]`

Indexing: Same as string `["a", "b", "c"][1]` gives "b"

```
append(newitem), insert(index, item), remove(item), pop(),
reverse(), len(mylist)
```

is vs ==

`is` checks if the object are the same in memory!

`==` is for checking if the objects have the same value

```
A = [1,2]

B = [1,2]

A is B # False

A == B # True
```

Creating Dictonary

- used to store data values in key:value pairs

Dictionary Operations

get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
values()	Returns a list of all the values in the dictionary
pop()	Removes the element with the specified key

Other Dictionary Operations

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
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(classmethod)

Creating set

Set operations

`x1 | x2` or `x1.union(x2)`: Union

`x1.intersection(x2)`: Intersection

`x1.isdisjoint(x2)`

`x1.issubset(x2)`

`x1.issuperset(x2)`

Set operations

```
x.add(<elem>) # add item to the set

x.remove(<elem>) # Raise error if element not already in set

x.pop() # Remove random items from the list

x.clear() : make the set empty
```

Functions & Scope

Other than built in functions like print(), input(), len() etc, we have user defined functions

User defined functions:

- Block of code that runs only when called upon.
- Pass data and parameters
- Function returns result

```
def function_name(parameters):
        <statements>
        return value

returned_value = function_name(arguments)
```

```
def my_function():
  print("Hello from a function")

my_function()
```

```
def fellowship(int a)
        print(a)
        a = a+1
        return a

num = 5
value = fellowship(num)
print(value)
```

Functions & Scope contd…

Functions are reusable code

Syntax

```
def add(a, b=1):
    return a + b
```

Called using

```
add(2,4)
add(2)
add(a=4,b=6)
```

Functions & Scope contd…

For Functions and its scope, there's **L.E.G.B** rule

- **L**ocal scope

  A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

- **E**nclosing (or nonlocal) scope

  As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

- **G**lobal scope

  A variable created in the main body of the Python code is a global variable and belongs to the global scope.

  Global variables are available from within any scope, global and local.

- **B**uilt-in scope

  It's the scope where essentially all of Python's top level functions are defined, such as len, range and print.

  When a variable is not found in the local, enclosing or global scope, Python looks for it in the builtins.

Functions & Scope contd…

LEGB code

Functions & Scope contd…

Global Keyword:

If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.

```
x = 200

def test_func():
      global x
       x = 500

test_func()
print(x)
```

Pass:

The pass statement is used as a placeholder for future code.

Nothing happens when the pass statement is performed, but you **avoid error** receiving when empty code is prohibited.

In loop declarations, function definitions, class definitions, or if statements, no empty code is permitted.

```
def myfunction():
      pass
```

Functions & Scope contd…

Positional arguments:

The argument should be in correct position in function call.

```
call_func(arg1, arg2, arg3)
```

```
def info(name, age):
      print(f"Hi, my name is {name}. I am {age * 365.0}
       days old.")


info("Soy", 22.0)

info(22.0, "Soy")   #ERROR
```

Keyword arguments:

Position does not matter as the values are labelled with keyword.

```
call_func(arg_name = arg1)
```

```
def info(name, age):
      print(f"Hi, my name is {name}. I am {age * 365.0}
       days old.")


info(name = "Soy", age = 22.0)

info(age = 22.0, name = "Soy")   #Works fine
```

END OF DAY-1

Comprehension

List comprehension

- Most distinctive aspect of python.
- Single line of code to construct powerful functionality.

Syntax:

```
newList = [ expression(element) for element in oldList if condition ]
```

List comprehension Vs For Loop

```
# Using for loop
test_list = []

for each in 'Software fellowship!':
    test_list.append(character)

# Display list
print(test_list)
```

```
# Using list comprehension to iterate through loop
test_list = [each for each in 'Software fellowship!']

# Displaying list
print(test_list)
```

Comprehension contd…

Dictionary comprehension

- Similar to List comprehension

Syntax:

```
{key: value for (key, value) in iterable}
```

```
# Lists to represent keys and values
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]

# but this line shows dict comprehension here
myDict = { k:v for (k,v) in zip(keys, values)}

# Alternative
# myDict = dict(zip(keys, values))

print(myDict)
```

Comprehension contd…

Generator syntax using comprehension

- Also similar to List comprehension
- **Does not** construct List object
- Generates the **next** element in demand

```
# List Comprehension
list_comprehension = [i for i in range(11) if i % 2 == 0]

print(list_comprehension)
```

```
# Generator Expression
generator_expression = (i for i in range(11) if i % 2 == 0)

print(generator_expression) # shows error

generator_expression.next() # 2
generator_expression.next() # 4
…. and so on
```

```
# to print out generator expression
for item in generator_expression:
    print(item)
```

Comprehension contd…

Knowledge Check

Q1. Display square of numbers from 1 to 10.

Q2. Toggle case of each character in a string.

Q3. Reverse each string in a tuple.

Q4. Display the sum of digits of all the odd elements in a list.

File handling

```
f = open(filename, mode, encoding='utf-8')

# mode: what to do with the file

# encoding: how to interpret the file

f.read() # read the whole file at once

f.readlines() # read the whole file as a list of lines' content

f.write() # write the string to file

f.writelines() # write the list of strings to file
```

File handling

Options for mode

Mode	Description
`'r'`	Open a file for reading. (default)
`'w'`	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
`'x'`	Open a file for exclusive creation. If the file already exists, the operation fails.
`'a'`	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
`'t'`	Open in text mode. (default)
`'b'`	Open in binary mode.
`'+'`	Open a file for updating (reading and writing)

Source: programmiz

File handling (Context Managers)

```
class ContextManager:
    def __init__(self):
        print('init method called')
    def __enter__(self):
        print('enter method called')
        return self
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('exit method called')
with ContextManager() as manager:
    print('with statement block')
```

We will come back to this!

Mutable vs Immutable Objects

list, dictionary, set, user-defined classes are mutable

All other objects are immutable

Example:

```
A = 10 # A points to 10

A += 20 # A points to 30
```

If you add 20 to 10, the old 10 is discarded and a new 30 is made

This is unlike other programming languages where 20 is modified in memory.

But if you say modify a list, its modified in place

```
A = [1, 2, 3]

A[1] = 5 # A = [1, 5, 3]
```

Mutable vs Immutable Objects contd…

Mutable objects:

list, dict, set, byte array

Immutable objects:

int, float, complex, string, tuple, frozen set [note: immutable version of set], bytes

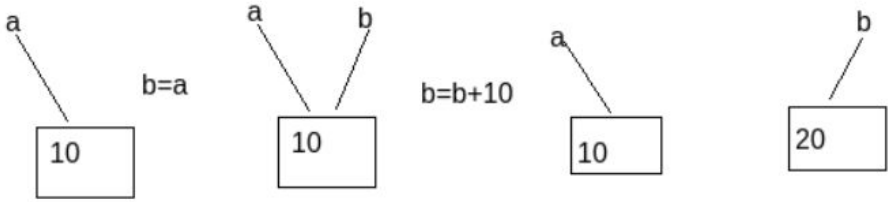
```
# Python code to test that
# tuples are immutable


tuple1 = (0, 1, 2, 3)
tuple1[0] = 4 //ERROR

print(tuple1)
```

```
# Python code to test that
# lists are mutable

color = ["red", "blue", "green"]
print(color)

color[0] = "pink"
color[-1] = "orange"
print(color)
```

Values vs References

- Value type:
  - All numeric type, boolean, string are value type in python

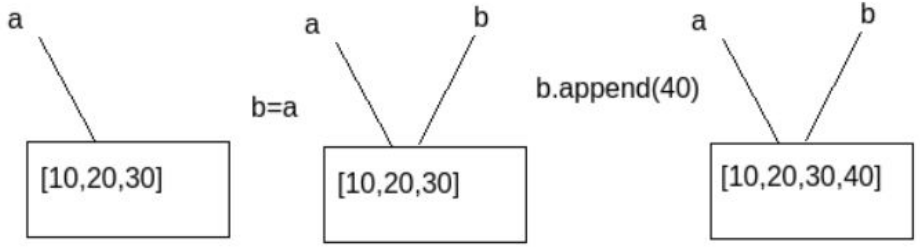
```
a = 10
b = a
b = b+10
print(a) # prints 10
print(b) # prints 20
```



Values vs References

- A reference type stores a reference to its data.
  - List, tuple, dict, set

```
a = [10,20,30]
b = a
print(a)  # prints [10,20,30]
b.append(40)
print(a)  # prints [10,20,30,40]
print(b)  # prints [10,20,30,40]
```



How to copy a list?

```
a = [1,2,3,4]

b = a # Remember: list variable is a reference variable, any future change in b also changes values in a

# one simple trick

b = a[:] #(recall: list slicing)
```

Classes and type

How to store information(feature:name, height, weight, hair_color,) of some student1 ???

You may think of

```
name = "John"
height = 5.9
weight = 55
hair_color = black
```

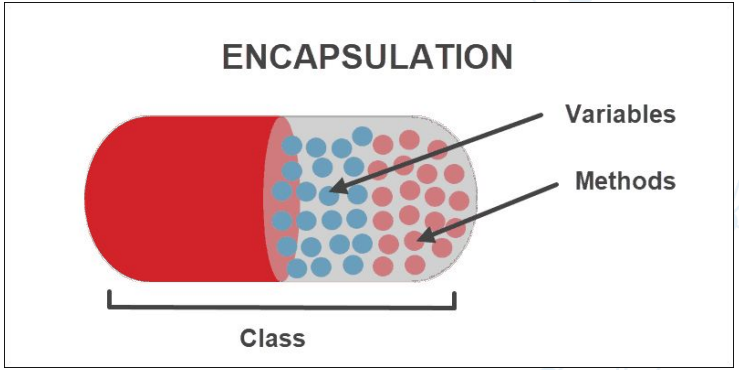
They seem to be related data, why not to represent them in a single unit/entity? This is where the concept of class(object oriented programming) emerges.

Classes and type

Easy way!

Encapsulated everything into a single entity(aka object)

- Variables: the attributed
- Method : the functionality object offers



Classes and type

Let’s formalize class in a pythonic way:

```
class Person(object):
    """A simple class."""           # docstring
    species = "Homo Sapiens"        # class attribute


    def __init__(self, name):       #special method
        """This is the initializer. It's a special
           method (see below).
        """
        self.name = name            #instance attribute
```

```
def __str__(self):          # special method
    """This method is run when Python tries
       to cast the object to a string. Return
       this string when using print(), etc.
    """
    return self.name

def rename(self, renamed):    # regular method
    """Reassign and print the name attribute."""
    self.name = renamed
    print("Now my name is {}".format(self.name))
```

Special methods in class

Some of the class's methods have the following form: *__functionname__(self, other_stuff)* . All such

methods are called "magic or dunder methods" and are an important part of classes in Python.

`__init__()` method:

The method that is first run when you create a new object, or new instance of the class.

Attributes that apply to a specific instance of a class (an object) are called *instance attributes.* They are generally defined inside `__init__()` ; this is not necessary, but it is recommended.

Operator Overloading

Operator overloading in Python is implemented with magic methods.

```
__add__ : +
__mul__ : *
__truediv__ : /
__sub__ : -
__pow__ : **
```

```
__str__:
__concat__ : string1 + string2
__lt__ : <
__or__ : or
```

Classes and type

`__str__():`

- To compute nicely printable string representation of an object.
- The return value must be a *string* object.
- Used mainly for creating end-user output

`__repr__():`

- a representation that has all information about the object
- Used mainly for development and debugging

Python is a duck typed language

Python: A duck typed language

“If it walks like a duck, and it quacks like a duck, then it must be a duck.”

```
class Duck:
    def __init__(self, name):
        self.name = name
    def quack(self):
        print('Quack!')


class Car:
    def __init__(self, model):
        self.model = model

    def quack(self):
        print('I can quack, too!')
```

Since Python is a dynamically typed language, we don’t have to specify the data type of the input arguments in a function.

```
def quacks(obj):
    obj.quack()
```

```
>>> donald = Duck('Donald Duck')
>>> car = Car('Tesla')
>>>
>>> quacks(donald)
'Quack!'
>>>
>>> quacks(car)
'I can quack, too!'
```

Inheritance

Say you have a car class that has a lot of attributes and methods

Now when you need a Lamborghini class do you rewrite the same things again?

Answer: No, Use inheritance

We make Lamborghini inherit from the Car class.

Then all methods and attributes of the Car class are available in the Lamborghini class.

Inheritance

Here's how to inherit

```
class Car:
      def make_noise():
            print("vroom")

class Lamborghini(Car):
      pass

L = Lamborghini()
L.make_noise() # vroom
```

Inheritance

You can inherit from multiple base classes

```
class Car:
    def make_noise(self):
        print("vroom")

    def same_name(self):
        print("sn car")

class Automobile:
    def pay_taxes(self):
        print("taxes")

    def same_name(self):
        print("sn mobile")

class Lamborghini(Car, Automobile):
    def pay_taxes(self):
        print("lambos pay special tax")
```

```
L = Lamborghini()
v = L.make_noise()  # vroom
t = L.pay_taxes()  # lambos pay special tax
r = L.same_name()  # sn car ; why? See MRO
```

Methods

A class can have three types of method defined

@staticmethod

@classmethod

regular method

```
class MyClass:
        def regular_method(self):
                pass
        @staticmethod
        def static_method():
                pass
        @classmethod
        def class_method(cls):
                pass
```

Here's a good video on this

https://www.youtube.com/watch?v=SXApHXsDe8I

Methods

Invoke a regular method

```
object.regular_method()
```

A static method is related to the class as a whole rather than a specific object

Invoke a static method

```
Classname.static_method()
```

But `object.static_method()` will work as well

Methods

Classmethod get the class parameter!

Mostly used for alternative constructors

Useful in inheritance:

Dog inherits Animal

Dog.from_json() # will create Dog

Animal.from_json() # will create Animal

```
class Calendar:
        def regular_method(self):
                pass
        @classmethod
        def from_json(cls):
                c = cls()
                return c
```

Errors and Exceptions

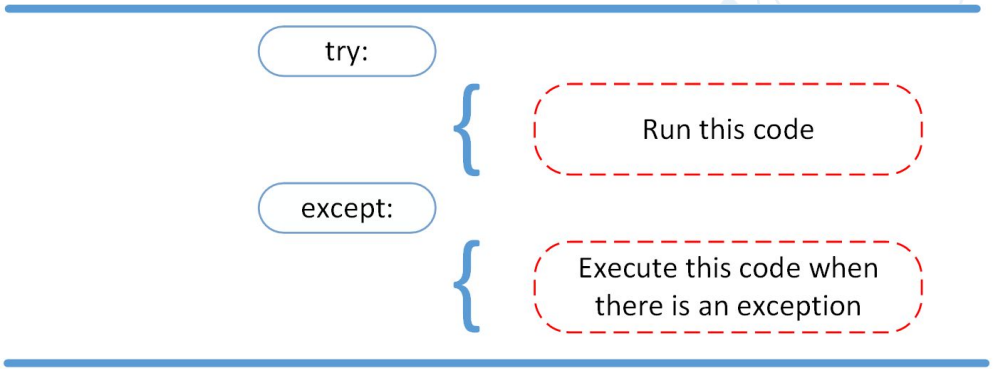
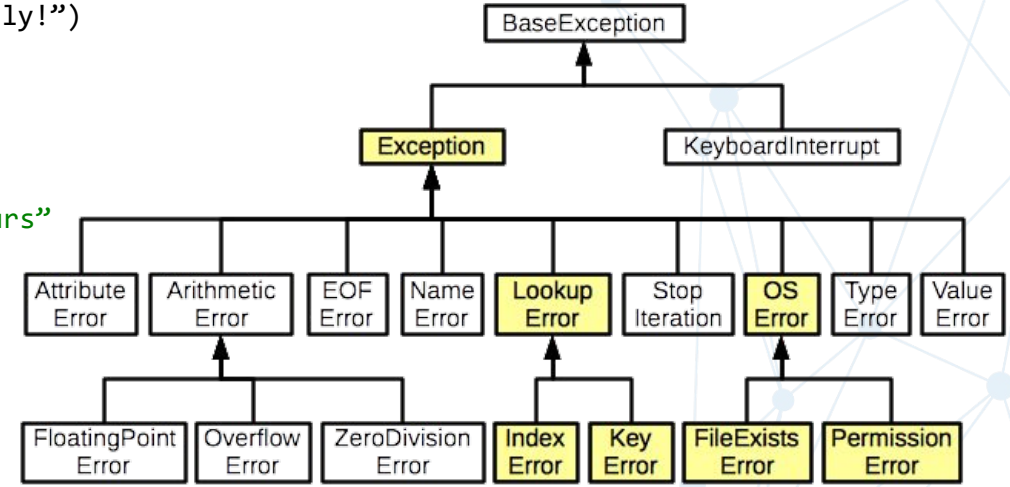
Syntax errors:

- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                     ^
SyntaxError: invalid syntax
```

- Such type of errors occur when the syntax(grammar) specified on the language do not match, thus the interpreter starts yelling at you.

Exceptions

- Exceptions are exception to the program flow(🤪) that can never be predicted.
- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal:
- ```
  def division(a,b)
        return a/b
  division(5,2) # returns 2.5
  division(5,0) # ZeroDivisionError: division by zero
  ```
  ```

Exceptions handling

- When an exception occurs, it interrupts the flow of the program. If the program can handle and process the exception, it may continue running. If an exception is not handled, the program may be forced to quit.
- How to handle such exceptions???

- Use try except statement

  - If any exceptions occurs inside the `try` block, the program execution flow follows the `except` block



https://realpython.com/python-exceptions/#:~:text=The%20try%20and%20except%20block%20in%20Python%20is%20used%20to,in%20the%20preceding%20try%20clause.

Exception handling(contd.)

- ```
  def division(a,b)
        try:
              result = a/b
              print("result computed successfully!")
              return result
        except:
              print("Division by Zero occurs")
  division(5,2) # returns 2.5
  division(5,0) # prints "Division by Zero occurs"
  ```

- `a/b` inside the `try` block throws an exception `ZeroDivisonError`, then the program execution flow follows the `except` block

Types of exceptions


  ```

Context Managers

Context managers make a resource available within a certain scope

A context manager has life cycle methods! __enter__ and __exit__

Context manager support special syntax

They have error capturing mechanism

```
with ctx as alias:
    Do something with alias
```

Context Managers

```
class ContextManager:
    def __init__(self):
        print('init method called')
    def __enter__(self):
        print('enter method called')
        return self # whatever we return here becomes the alias
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('exit method called')
with ContextManager() as manager:
    print('with statement block')
```

Everything is object

In python everything is an object!

Even functions are objects!

```
def add(a,b):
     return a = b
type(add) # function object
```

Even classes are object themselves

All classes are instances of a class called "type". The type object is also an instance of type class. `isinstance(YourObject, type) # True`

The `type` class is used to instantiate other classes `type(name, bases, dict)`

Lambdas

Lambdas are mini functions

One-off functions

Syntax: `lambda x: return_value`

Example: `lambda x: x * x`

Higher Order Functions

Function which take functions as arguments

```
map(function, iterable) # apply function to each element of iterable
```

```
filter(function, iterable) # filter each element of iterable using function
```

Higher Order Functions

```
def square(a):
    return a ** 2


l = [ 1, 2, 3]
list(map(square, l)) # [1, 4, 9]
list(filter(lambda x: x % 2 == 0, l)) # [2]
```

END OF DAY 2

Unpacking

- extract the values back into variables
- For tuple, list

```
fruits = ("apple", "banana", "cherry") #packing

(green, yellow, red) = fruits #unpacking
```

Unpacking with *

- If length of variable and values do not match

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits #Guess the output?

(green,*yellow,red)= fruits #try this….
```

Function argument with *args and **kwargs

- When we don't know exact number of positional argument up front **`*args`** can be really useful, because it allows you to pass a varying number of positional arguments in the function. Variable `args` then stores arguments in **tuple***(Recall: tuple → (1,2,3,4,5)).*
- Similarly, **`*kwargs`** variable accepts keyword (or named) arguments and stores in the form of **dictionary***(recall: dictionary → {"first":1, "second":2}).*

Parameter	Details
`arg1, ..., argN`	Regular arguments
`*args`	Unnamed positional arguments
`kw1, ..., kwN`	Keyword-only arguments
`**kwargs`	The rest of keyword arguments

Function argument with *args and **kwargs(contd.)

What we used to do

```
# sum_integers_args.py
def my_sum(first, second, third):
    # add all numbers
     result = first + second + third
     return result

print(my_sum(1, 2, 3))
```

Can also be done as

```
# sum_integers_args.py
def my_sum(*args):
    result = 0
    # Iterating over the Python args tuple
    for x in args:
        result += x
    return result

print(my_sum(1, 2, 3))
```

Function argument with *args and **kwargs(contd.)

What we used to do

```
# concatenate.py

def concatenate(a, b, c):

    result = ""

    result = a + b + c

    return result


print(concatenate(a="Hello",
b="Python", c="geeks")

#returns HelloPythongeeks
```

Can also be done as

```
# concatenate.py

def concatenate(**kwargs):

    result = ""

    # Iterating over the Python kwargs
dictionary

    for arg in kwargs.values():

        result += arg

    return result

concatenate(a="Hello", b="Python", c= "geeks")

#returns HelloPythongeeks
```

Args only/Kwargs only syntax

- `fn_name(positional_or_keyword_parameters, *, keyword_only_parameters)`

  This allows us to pass `positional_or_keyword_parameters` and `keyword_only_parameters`, separated by `*`

```
def fellowship(first, second, *, third):
      pass
fellowship(1,2,3) # TypeError: f() takes 2 positional arguments but 3 were given
fellowship(1,2,third=3) # OK
fellowship(1,second=2,third=3) # OK
```

- `fn_name(positional_only_parameters, /, positional_or_keyword_parameters,*, keyword_only_parameters)`

  This allows us to pass `positional_only_parameters, positional_or_keyword_parameters`,separated by / and `keyword_only_parameters` separated by `*`.

```
def fellowship(first,/, second, *, third):
      pass
fellowship(1,2,3) # TypeError: f() takes 2 positional arguments but 3 were given
fellowship(1,2,third=3) # OK
fellowship(1,second=2,third=3) # OK
fellowship(first=1,2,third=3) # SyntaxError: positional argument follows keyword argument
```

Inner Functions

- Function inside function

```
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")

    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()
parent()
```

Function Pointer

```
def myfun():
    print("Hello from function")
myfun()
fun_p = myfun #notice, parenthesis is not used
Now myfun() can also be called by using fun_p
fun_p()
```

Decorator

- Function Decorator: A function that takes another function as argument and return yet another function
- Class Decorator: A function that takes a class and returns another class

Simple Decorator

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
def say_whee():
    print("Whee!")
say_whee = my_decorator(say_whee)
say_whee()
```

Can you guess the output?

Syntactic Sugar for Decorator

- A simple way to make code little less clunky is to use @ symbol

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
@my_decorator
def say_whee():
    print("Whee!")
```

iter and next

iter and next are built in functions

Iterator : A pointer to a element in iterable

Iterable : A object that defines __iter__

For example: list, dictionary, (you can create your own object with __iter__ method)

iter : takes a iterable and returns a iterator

next : takes a iterator and returns whatever the iterator is pointing (throws an error if iteration is complete)

Another syntax with default if iteration is complete: next(iterator, default)

Generators

Generators are a special kind of iterators

Generators are functions with the yield keyword instead of return

The function will now return generator instead of the return value

The generator can be iterated through just like with iterator

yield : return for now but I will be back so remember me

Imports

import re

from package.module import function_name as new_name

Module: A python file

Package : collection of modules (a folder)

Environment Variables

PYTHONHOME : Search here for standard libraries

PYTHONPATH : Search for imported stuff here too, search here first

See your import locations at sys.path

sys.path is a list, its mutable, python program can change import locations at runtime

Imports

from functors import partial

fn_name = partial(fn, *args, **kwargs)

```
def raise(x, y):
    return x**y

from functors import partial
raise_to_three = partial(raise, y=3)
```

Debugging Python Code

DEMO using VScode

Pdb is another option, comes bundled with the standard library

Gives you interactive python shell for debugging.

For more info: https://docs.python.org/3/library/pdb.html

Pip and venv

PyPI : Python Package Index is a repository of many python packages

Pip is the cli tool for installing, uninstalling packages

pip install django

pip uninstall django

pip freeze

pip install -r requirements.txt

Advanced tools: pyenv, poetry, pipenv

Pip and venv

Colliding dependencies

- Application A needs Django 3
- Application B needs Django 4

What to do when you import django?

Solution: virtual environment

Pip and venv

A virtual environment is isolated environment taylored for a specific application

python3 -m venv <virtualenv_name>

source <virtualenv_name>/bin/activate

deactivate

How does it work? It changes the PATH and creates a different location to install packages

Advanced tools: pyenv, poetry, pipenv

Simple scraper

Now we will use the requests module to extract a data from a simple website

Using only string operations

Python jargons (Have a look on your own)

Alias

Benevolent Dictator For Life(BDFL)

Cheese shop

Deep copy / shallow copy

Eager / lazy

higher-order function

Pythonic

More things to learn

Metaclasses

Monkey patching

Named operators (is, in)

Garbage collection

Currying, Closure

Itertools

Advanced Builtins

super()

Interning

More Python please!

https://sadh.life/post/builtins/

Pluralsight, Robert Smallshire

Real Python (https://realpython.com)

Freecodecamp

Clean coding (Uncle Bob: https://www.youtube.com/playlist?list=PLmmYSbUCWJ4x1GO839azG_BBw8rkh-zOj)