

Python: the Language

Python Introduction

**Datatypes and
Variables**

Lists and loops

Selection

Functions

Organise code & IO

Python Introduction

- The philosophy of Python
 - Programming is not only easy to write but also easy to read.
- Python is *simple* and *small*.
- Python is one of the most popular programming languages that has survived the test of time.

- Python is one of the three languages that students must master
 - Java (C#/C++) : Server/High-performance
 - Python: AI/LLM and SE
 - JavaScript: App development

Static and Dynamic Language

Program creation has two steps:

1. **Compilation** – translate source code into low-level code.
2. **Execution** – run the translated low-level code.

Compilation

- In this step, a static programming language such as Java or C# checks types to avoid errors.
 - Static languages have types.

```
// Java compiler (javac) finds an error at compile time  
String hello = 123;  
hello[10]; // This error is not possible
```

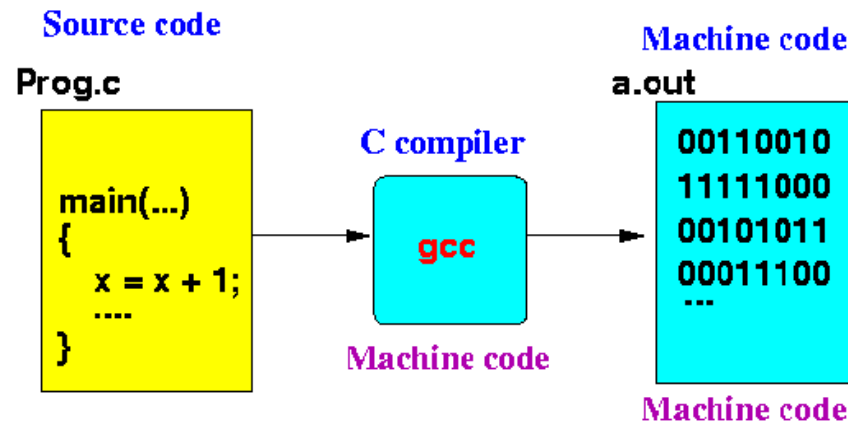
- Python (a Dynamic programming language) does not check types.
 - This may cause a serious bug that is hard to find.
 - The type bug will appear only when we run it.

```
hello = "hello"; hello = 40  
hello[10] // This error will happen at runtime
```

Byte code conversion

- In the compilation step, modern programming languages translate high-level code into low-level code.
 - Humans understand high-level code.
 - Machine understands low-level code.

- System programming languages, such as C/C++/Rust, compile high-level code directly to machine code.
- CPU can execute the machine code directly.



- Java or Python translates high-level code into bytecode.
- CPU cannot execute bytecode directly, so we need a VM (Virtual Machine).

Java:	Bytecode:
1 public class Example{	1 public class Example{
2	2
3 int x;	3 i x
4	4
5 static int y;	5 static i y
6	6
7 public int getX() {	7 public getX()i
8 return x;	8 aload 0
9 }	9 getfield Example.x : i
10	10 ireturn
11	11
12 public void setX(int x) {	12 public setX(i)V
13 this.x = x;	13 aload 0
14 }	14 iload 1
15	15 putfield Example.x : i
16	16 return
17	17
18 public int increaseY() {	18 public increaseY()i
19 return y++;	19 getstatic Example.y : i
20 }	20 dup
21	21 iconst_1
22	22 iadd
23	23 putstatic Example.y : i
24	24 ireturn
25	25
26 public void EventHandler() {	26 public EventHandler()V
27 setX(1);	27 aload 0
28 }	28 iconst_1
29	29 invokevirtual Example.setX()V
30	30
31 }	31 }

Execution

- Java's JVM uses JIT to compile bytecode to machine code during execution.
- Python executes bytecode line by line without compilation.
- Python runs 10–100 times slower than Java.

Python: Pros and Cons

Pros of Python

- Small and easy to learn.
- Fast compile time without type checking.
- Very high-level language.
- Short build-debug-rewrite cycle.
- Many packages and tools.

Cons of Python

- No type checking, so it's hard to find bugs.
- Type hinting is introduced and widely used.

```
age: int = 20  
name: str = "Alice"  
is_active: bool = True
```

```
def greet(name: str) -> str:  
    return f"Hello, {name}"
```

Slow execution time (No JIT)

- Mojo can be up to 35,000 times faster with direct machine code.
- PyPy is at least 7 times faster using JIT, no code changes.
- Python 3.13 adds experimental JIT support.

Datatypes and Variables

- Python has datatypes and variables.
- Python does not check types for variables.

```
# x can reference any time without checking  
x = 10 # 10 is an int type value  
x = Hello # "Hello" is a string type value
```

- A datatype (at a high level) is a set of specific values and their properties.
- The number 15 is an integer datatype, and the integer datatype has properties such as '+'.

- String datatype also has a '+' property, but the action differs (this is called Polymorphism).

```
x = 10; y = 20;  
z = x + y # z = 30
```

```
a = "Hello"; b = "World";  
c = a + b # c = "HelloWorld"
```

- We can use `type()` function to check the type name

```
type("Amsterdam") # -> <class 'str'>
type(1) # -> <class 'int'>
type([1,2,3]) # -> <class 'list'>
type({'a':10}) # -> <class 'dict'>
{type({1,2}) # -> <class 'set'>
type(true) # -> <class 'bool'>
```

Expression

A Python expression means computation.

- We can use the `+` symbol to add two values.
- We have other symbols such as `-`, `*`, `/`, `'` to compute.

```
print(3 + 4)
print(3 - 4)
print(3 * 4)
print(3 / 4) # returns 0.75 (double type value)
print(3 // 4) # returns 0 (int type value)
print(10 + 20*4) # 90 as multiplication first
print((10+20)*4) # 120
```

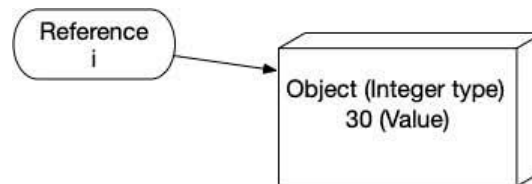
Variables

Python is pure OOP; everything is an object.

- No need for `new` to create objects.
- Python code is simple.

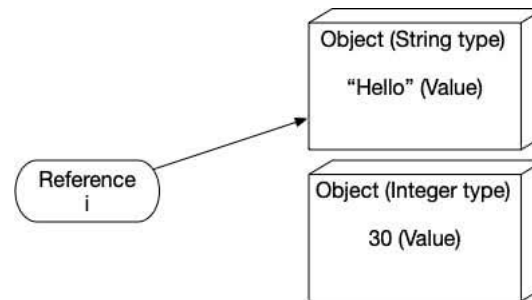
```
Integer i = new Integer(30); //Java  
i = 30 // Python
```

- The Variables in Python are references.
- The value assignment is an instantiation and reference assignment.



```
i = int(30) # 30 is an int  
i = 30 # same
```

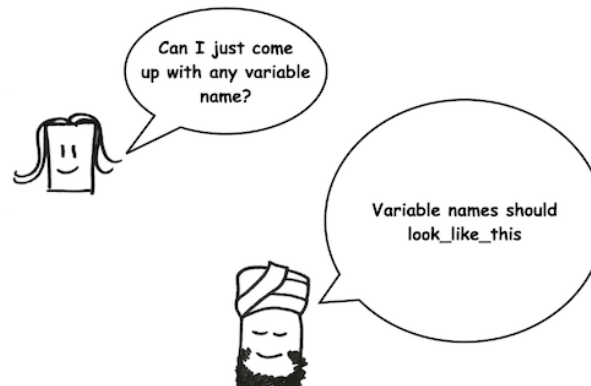
- In this example, 30 is an integer object, and variable i can reference any object freely.



```
i = 30  
i = "Hello"
```


Naming rules

- Class starts with a capital letter.
- It should not start with a number or special characters
- No spaces; use `_` instead.



Floating point number

- There is an infinite number of floating-point numbers.
- So, we approximate it.

$$\frac{2}{3} = 0,66666666...7$$


```
>>> 1.2 + 1.0
2.2
>>> 1.2 - 1.0
0.199999999999999999996 # surprise!
```

String format and interpolation

Python provides an f-string (format string).

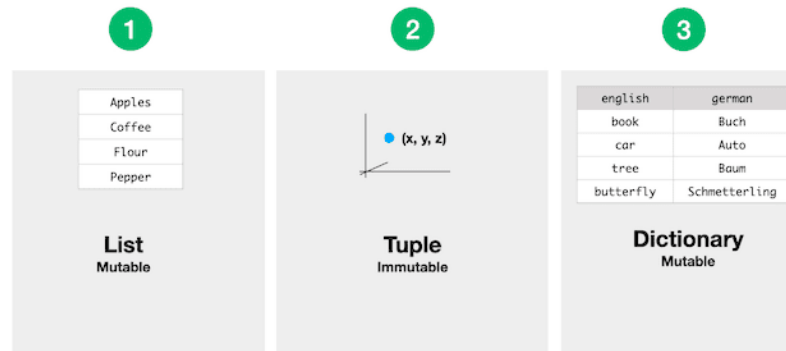
- It is a way of string interpolation (use an expression in a string).

```
name = "world"  
print(f"Hello {name}") # prints out "Hello world"  
print("Hello " + "world") # alternative solution
```

Lists and loops

Python has three lists.

- A list (mutable).
- A tuple (immutable).
- A dictionary (list with keys).



A List

- We can use the `[...]` symbol or the `list` function to make a list.
- We use the `append` method to add an item to a list.

```
names = ["Vera", "Chuck", "Dave"]  
names = list() # alternative solution  
names.append("Vera"); ...
```

- We use an index to access a list.
- Python supports for-in-loop.

```
names = ["Vera", "Chuck", "Dave"]  
print(names[0]) # "Vera"  
for name in names:  
    print(name)
```

- Python uses `len()` to get an array's size, not `names.length()`.
- This relies on duck typing: `len()` calls the list's hidden `__len__` method.

```
names = ["Vera", "Chuck", "Dave"]  
len(names) # 3  
names.__len__() # same as before
```

- Python has built-in functions, such as `len()` and `print()`.
- There are many built-in functions.

		Built-in Functions		
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>

Tuples

- Tuples are immutable lists.
- Created with the `()`, accessed with the `[]` operator.

```
x = (1, 2, 'a')  
# AttributeError:  
# 'tuple' object has no attribute 'append'  
x.append(4)  
print(x[0])
```


- Tuples have a `__len__` method, so the built-in `len()` function works on them thanks to duck typing.

```
x = (1, 2, 'a')  
print(len(x)) # 3  
x.__len__() # 3
```

Dictionary

- A Python dictionary is a key/value pair.
- Created with `{...}`.

```
x = {'a':10, 'b':20}
print(x['a'])
x.update({'c':30})
for i in x: print(i) # print keys
for i in x: print(x[i]) # print values
```

- Behind the scenes, a list is a dictionary with keys as an index.

```
x = [1,2,3]  
x = {0:1, 1:2, 2:3}
```

- The `in` operator checks if a key is in the dictionary.

```
x = {'a':10}
if 'b' in x: x['b'] += 20 # update the value
else x['b'] = 20 # create a new
```

Selection

- Relational Expression
- If statement

Relational Expression

- Python has relational operators: `>`, `<`, `==`, `!=`, `>=`, `<=`.
- Logical operators `'and'`, `'or'`, `'not'` combine expressions.

```
5 > 4 # True
```

```
5 > 4 and 5 > 6 # False
```

- Python uses `==` to compare strings (checks value equality).

```
"Jim" == "Sam"    # Like Java's .equals()
```

If Statement

- Python does not have a switch/case statement.
- Python supports only if statements

```
if x == 3: print("x is 3")  
elif x == 4: print("x is 4")  
else: print("Not")
```


If expression

- Python supports if expressions.

```
x = 3 if value == 10 else 20 # python  
x = value == 10 ? 3 : 20;    // Java
```

Functions

Python supports functions.

- Define a function with `def`, a name, parameters in parentheses, and a body.
- Use `return` to send back a value from the function.

- We can call the function by giving arguments that match the parameters.

```
def add(x, y):  
    return (x + y)  
print(add(1,2))
```

```
def add(x = 1, y = 2): # default argument  
    return (x + y)  
print(add()) # same
```

Nested Functions

- Python supports nested functions (functions in a function).

```
def complex_add(x,y):  
    def add(x, y): return (x + y)  
    return add(x,y) + add(x,y)  
complex_add(1,2) # add(1,2) + add(1,2) -> 1+2+1+2  
complex_add(y=2, x=1)
```

Closure

- Nested functions can access variables from their outer function –this is called a closure.
- Closures let inner functions use those variables even after the outer function ends.

```
def outer():  
    message = "Hello, world!"  
    def inner():  
        print(message)  
    return inner  
  
greet = outer()  
greet()  # Output: Hello, world!
```

- `inner` accesses `message` from `outer`, forming a closure.
- `greet` retains `message` even after `outer` ends.

Explode arguments

- Use `*` to unpack a list into positional arguments.
- Use `**` to unpack a dict into named arguments.

- Explode arguments make code simple and easy to read.

```
x = [1,2]
print(add(*x)) # add(1, 2)
y = {'x':10, 'y':20}
print(add(**y)) # add(x = 10, y = 20)
```


Organize code & IO

- File read/write
- Error Handling

File read/write

- We can use the `open()` method to open a file and read/write strings.

```
text = "Hello, this is a text file!"  
file = open ("hello.txt", "w") # "w" for write  
file.write(text)  
file.close()  
  
file = open ("hello.txt", "r")  
for str in file.readlines(): # read all lines in a string  
    print(str)
```

Error Handling

- When we make errors in coding, and Python detects them, it is a `compile-time error`.
- When an error occurs during the execution of the code, Python detects them: `run-time error` or `exceptions`.

```
print("Hell" # Compile time error

file = open ("hello2.txt", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'hello2.txt'
```

A runtime error or exception is worse than a compile-time error:

- It is harder to fix.
- Users find the bug, not the software engineer.

- Use `try/except` to handle runtime errors in Python.
- Use `with/as` to automatically manage resources and ensure cleanup, like closing files, even if exceptions occur.

```
try:
    file = open ("hello2.txt", "r")
except FileNotFoundError as e:
    print(f"{e}!!!")

with open("hello2.txt", "r") as f:
    f.readlines()
```