Python control

🐍

Overview

- Flow controls
- Functions
- Classes introduction

Flow controls

If statement

```
if_stmt ::=  "if" expression ":" suite
             ("elif" expression ":" suite)*
             ["else" ":" suite]
```

If statement

```
name = "Donald"

if name == "Donald":
    print("This is the Art")
```

```
This is the Art
```

If statement

Indentation is Python’s way of grouping statements

You have to type a tab *or* space(s) *for each indented line*

If statement

```
name = "Donald"

if (name == "Donald"): # ←-------- Bad
    print("This is the Art")
```

```
This is the Art
```

If statement

```
name = "Nothing"

if name == "Donald":
    print("This is the Art")
else:
    if name == "Linus":
        print("This is Linux")
    else:
        print("Nothing found")
```

Why is it bad ? 😓

```python
name = "Nothing"

if name == "Donald":
    print("This is the Art")
else:
    if name == "Linus":
        print("This is Linux")
    else:
        print("Nothing found")
```

If statement

```python
name = "Linus"

if name == "Donald":
    print("This is the Art")
elif name == "Linus":
    print("This is Linux")
```

```
This is Linux
```

If statement

```python
name = "Nothing"

if name == "Donald":
    print("This is the Art")
elif name == "Linus":
    print("This is Linux")
else:
    print("Nothing found")
```

```
Nothing found
```

Truthy / Falsy semantic

```
>>> lst = []

>>> bool(lst )
False

>>> if lst:
...     print("Not empty list")

>>> if not lst:
...     print("Empty list")
Empty list
```

Truthy / Falsy semantic

```
>>> lst = [[]]

>>> if lst:
...    print("Not empty list")
```

Truthy / Falsy semantic

```
>>> bool(0)
False
>>> bool(0.)
False
>>> bool(0j)
False
>>> bool({})
False
>>> bool('')
False
>>> bool(range(0))
False
>>> bool(())
False
>>> bool(None)
False
```

Truthy / Falsy semantic

```
>>> lst = []

>>> if len(lst) == 0: # Not pythonic way
...     ...

>>> if lst:
...     ...

>>> if not lst:
...     ...
```

Chaining

```
>>> a, b, c, d = 10, 20, 30, 40

>>> a < b < c < d
True
```

Chaining

```
>>> a, b, c = 10, 20, 30

>>> a > b < c
False
```

Chaining

```
>>> a, b, c, d = 10, 20, 30, 40

>>> a > b < c < {}['key']
False

>>> a, b, c = 20, 10, 30

>>> a > b < c < {}['key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'key'

'key'
```

Same with ‘or’ operator

```
>>> a, b, c = True, False, False

>>> a or b or c or {}['key']
True

>>> a, b, c = False, False, False

>>> a or b or c or {}['key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'key'

'key'
```

Conditional expression (ternary operator)

```
>>> if True if 1 else False:
...     print(1)
# ???
```

Conditional expression
(ternary operator)

```
<expression1> if <condition> else <expression2>
```

Conditional expression (ternary operator)

The expression `x if C else y` first evaluates the condition, *C* rather than *x*. If *C* is true, *x* is evaluated and its value is returned;

otherwise, *y* is evaluated and its value is returned.

Conditional expression (ternary operator)

The expression `x if C else y` first evaluates the condition, *C* rather than *x*. If *C* is true, *x* is evaluated and its value is returned;

otherwise, *y* is evaluated and its value is returned.

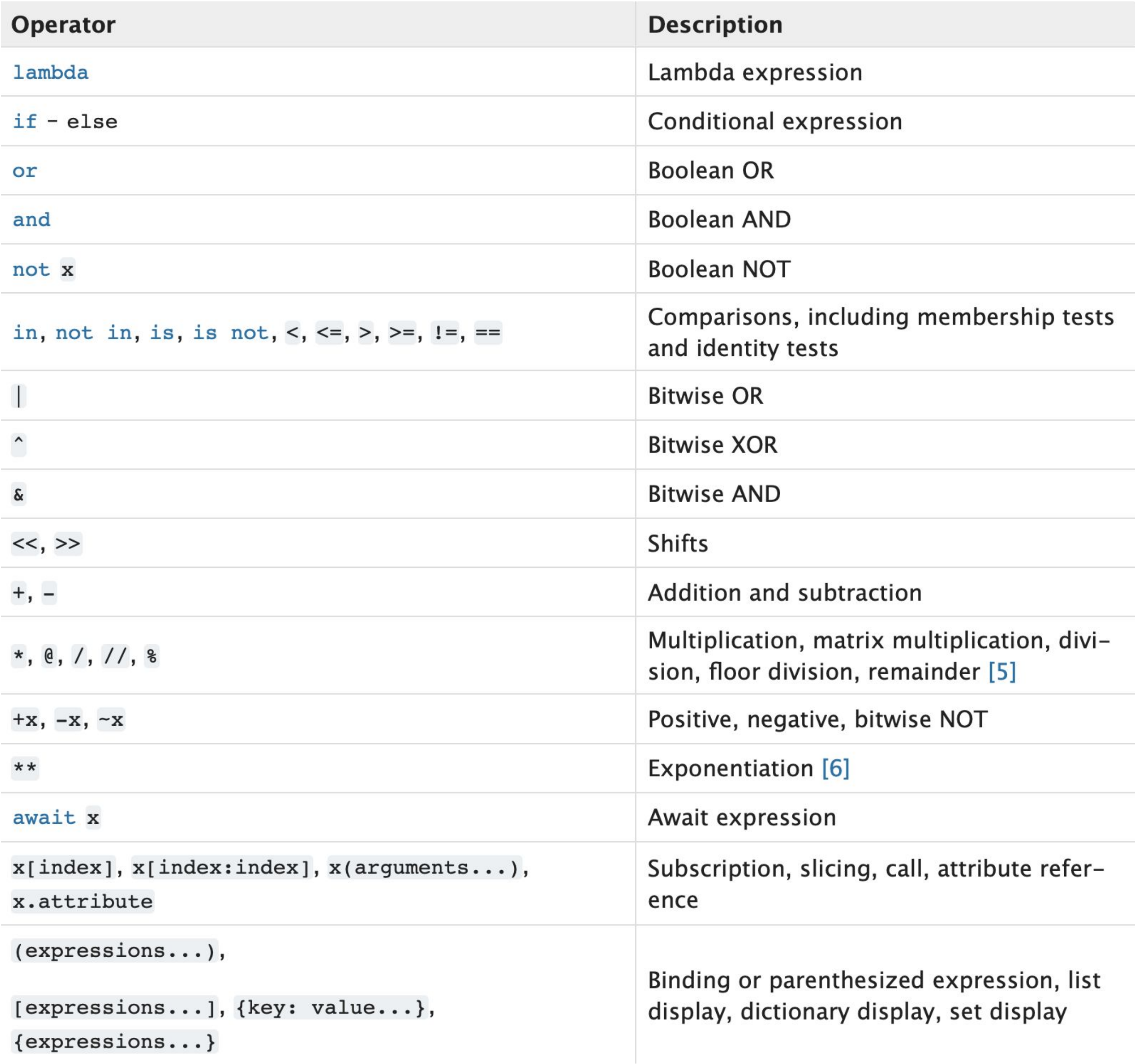
Conditional expression (ternary operator)

```
>>> {'hello': 1} if True else {}['hello']
{'hello': 1}
```

```
>>> {'hello': 1} if False else {}['hello']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'hello'

'hello'
```

Evaluation order



Operator	Description
lambda	Lambda expression
if – else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder [5]
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation [6]
await x	Await expression
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or parenthesized expression, list display, dictionary display, set display

while statement

```
>>> while True:
...     print("Inf loop")
Inf loop
Inf loop
Inf loop
Inf loop
Inf loop
Inf loop
Inf loop
Inf loop
Inf loop
Inf loop
...
```

while statement

```
>>> a = 0
>>> while a < 10:
...     print(a)
...     a += 1
0
1
2
3
4
5
6
7
8
9
```

while statement

```
>>> while True:
...     print(f"Enter {a!r}  loop")
...     if a > 5:
...         break
...     a += 1
Enter 0  loop
Enter 1  loop
Enter 2  loop
Enter 3  loop
Enter 4  loop
Enter 5  loop
Enter 6  loop
```

while statement

```
>>> while a < 5:
...     a += 1
...     print(f"Enter {a!r} loop")
...     if a % 2 == 0:
...         continue
...     print("a % 2 != 0")
Enter 1 loop
a % 2 != 0
Enter 2 loop
Enter 3 loop
a % 2 != 0
Enter 4 loop
Enter 5 loop
a % 2 != 0
```

while statement

```
>>> while a < 5:
...     if a == 6:
...         break
...     a += 1
... else:
...     print('a < 5')
a < 5

>>> while a < 7:
...     if a == 6:
...         break
...     a += 1
... else:
...     print('a < 5')
```

for statement

Rather than always iterating over an arithmetic progression of numbers, or giving the user the ability to define both the iteration step and halting condition (as C),

Python’s ***for*** statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

for statement

```
>>> bag_of_words = ['hello', 'my name is']
>>> for item in bag_of_words:
...     for char in item:
...         print(f"Char: {char!r}")
Char: 'h'
Char: 'e'
Char: 'l'
Char: 'l'
Char: 'o'
Char: 'm'
Char: 'y'
Char: ' '
Char: 'n'
Char: 'a'
Char: 'm'
Char: 'e'
Char: ' '
Char: 'i'
Char: 's'
```

for statement

```
>>> mapping = {'1': 1, '2': 2, '3': 3}

>>> for key, value in mapping.items():
...     print(f"{key!r}: {value!r}")

'1': 1
'2': 2
'3': 3
```

for statement

Iterating over a sequence does not implicitly make a copy

```
>>> for item in lst: # Never stops
...     lst.append(item + 1)
...     print(f"Item: {item}")

Item: 1
Item: 2
Item: 4
Item: 5
Item: 2
Item: 3
Item: 5
Item: 6
Item: 3
Item: 4
Item: 6
Item: 7
Item: 4
...
```

for statement

Iterating over a sequence does not implicitly make a copy

```
>>> lst = [1, 2, 4, 5]

>>> for item in lst[:]:
...     lst.append(item + 1)
...     print(f"Item: {item}")
Item: 1
Item: 2
Item: 4
Item: 5


>>> lst
[1, 2, 4, 5, 2, 3, 5, 6]
```

for statement

Iterating over a sequence does not implicitly make a copy

```
>>> lst = [1, 2, 4, 5]

>>> for item in lst[:]:
...     lst.append(item + 1)
...     print(f"Item: {item}")
Item: 1
Item: 2
Item: 4
Item: 5


>>> lst
[1, 2, 4, 5, 2, 3, 5, 6]
```

range()

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.

It generates arithmetic progressions.

range()

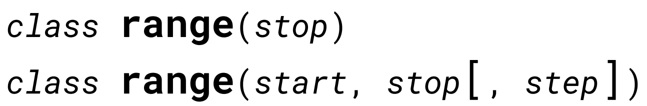
If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.

It generates arithmetic progressions.

range()

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```

range()



class **range**(*stop*)

class **range**(*start*, *stop*[, *step*])

The *range* type represents an **immutable** sequence of numbers and is commonly used for looping a specific number of times in *for* loops

range()

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

range()

```
>>> range_obj = range(10)
>>> range_obj
range(0, 10)

>>> 1 in range_obj
True

>>> range_obj[1]
1

>>> range_obj[::-1]
range(9, -1, -1)

>>> range_obj = range(10)

>>> list(range_obj[::-1])
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Bonus

```
>>> {range(10): 10, range(5): 5}

{range(0, 10): 10, range(0, 5): 5}
```

Functions

Functions

A function definition defines a user-defined function object

Functions

```
>>> def func():
...     """ Dummy function. Returns nothing """
...     pass


>>> type(func)
<class 'function'>

>>> func.__name__
'func'

>>> func.__doc__ # help(func)
' Dummy function. Returns nothing '

>>> hash(func)
8783939476151
```

Functions

```
>>> def sum(a, b):
...     return a + b


>>> sum(a=1, b=2)
3

>>> sum(1, b=2)
3
```

Functions

```
>>> sum.__annotations__
{}
```

Functions

Type annotation is not mandatory but recommended for readability purposes, built-in IDE lex-analyzers

…

For mypy is mandatory

```
>>> def sum(a: int, b: int) -> int:
...     return a + b
```

```
>>> sum.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class
'int'>}
```

Defaults

```
def student(first_name, last_name, grade=5):
    """ Function returns tuple with first name, last name and grade """
    return first_name, last_name, grade


>>> print(student("Mark", "Walters"))
('Mark', 'Walters', 5)

>>> print(student("Hugo", "Smith", 3))
('Hugo', 'Smith', 3)

>>> print(student(first_name="Hugo", last_name="Smith", 5))
Syntax Error: positional argument follows keyword argument (<input>, line
1)
```

Defaults

```python
def concat_and_multiply(lst_1, lst_2, number=1):
    """ Function concats two lists and multiply them by `number` """
    return (lst_1 + lst_2) * number
```

```
>>> concat_and_multiply([1, 2, 3, 4, 5], [11, 12, 1, 1, 1], 1)
[1, 2, 3, 4, 5, 11, 12, 1, 1, 1]
```

Defaults

```
def concat_and_multiply(lst_1, lst_2, *, number=1):
    """ Function concats two lists and multiply them by `number` """
    return (lst_1 + lst_2) * number
```

```
>>> concat_and_multiply([1, 2, 3, 4, 5], [11, 12, 1, 1, 1], 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: concat_and_multiply() takes 2 positional arguments but 3 were
given

concat_and_multiply() takes 2 positional arguments but 3 were given
```

Defaults

```
def concat_and_multiply(lst_1, lst_2, *, number=1):
    """ Function concats two lists and multiply them by `number` """
    return (lst_1 + lst_2) * number
```

```
>>> concat_and_multiply([1, 2, 3, 4, 5], [11, 12, 1, 1, 1], number=1)
[1, 2, 3, 4, 5, 11, 12, 1, 1, 1]
```

Defaults

```python
from typing import List

def append_to_list(element: int, lst: List = []) -> List:
    lst.append(element)
    return lst

>>> append_to_list(10)
[10]

>>> append_to_list(12)
[10, 12]
```

Defaults

```
from typing import List

def append_to_list(element: int, lst: List = []) -> List:
    lst.append(element)
    return lst

>>> append_to_list.__defaults__
([10, 12],)
```

*args, **kwargs

```
def sum(*args: int) -> int:
    result = 0
    for number in args:
        result += number
    return result

>>> sum(1, 2)
3
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, 4)
10
```

*args, **kwargs

```
def get_args_kwargs(a, b, *args, c, d, **kwargs):
    return args, kwargs
```

```
>>> get_args_kwargs(1, 2, 3, 4, 5, c=1, d=2, key='value')
((3, 4, 5), {'key': 'value'})
```

Call stack

```
def fib(n: int) -> int:
    """ Returns n th Fibonacci sequence element """
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

>>> fib(10)
55
```

Call stack

```
>>> fib(10000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in fib
  File "<stdin>", line 5, in fib
  File "<stdin>", line 5, in fib
  [Previous line repeated 2980 more times]
  File "<stdin>", line 3, in fib
RecursionError: maximum recursion depth exceeded in comparison

maximum recursion depth exceeded in comparison
```

Call stack

```
>>> import sys
... print(sys.getrecursionlimit())
3000

>>> sys.setrecursionlimit(100000)

>>> fib(10000)
KeyboardInterrupt
```

Call stack

```python
def fib(n: int) -> int:
    """ Returns n th Fibonacci sequence element """
    if n <= 1:
        return n

    a, b = 1, 1
    for i in range(2, n):
        c = a + b
        a, b = b, c
    return b
```

Call stack

```
>>> fib(10000)
3364476487643178326662161200510754331030214846068006390656476997
4680081442166662368155595513633734025582065332680836159373734790
4838652682630408924630564318873545443695598274916066020998841839
3386465273130008883026923567361313511757929743785441375213052050
4347701602264758318906527890855154366159582987279682987510631200
5754287834532155151038708182989697916131278562650331954871402142
8753269818796204693609787990035096230229102636813149319527563022
78376284415403605844025721143349611800230912082870460889...
```

Classes introduction