PyLearn-PyTutorial-4.ControlFlows

April 20, 2020

#

Learning Python

1 The Python Tutorial -> More Control Flow Tools

Link: https://docs.python.org/3/tutorial/controlflow.html

1.1 if Statements

[2]:

```
x = int(input("Please enter an integer: "))
if x < 0:
   x = 0
   print('Negative changed to zero')
elif x == 0:
   print('Zero')
elif x == 1:
   print('Single')
else:
   print('More than one')
```

```
Please enter an integer:  5
```

```
More than one
```

1.2 for Statements

[10]:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
  print(w, len(w))
```

```
cat 3
window 6
defenestrate 12
```

1.2.1 Strategy: Iterate over a copy

[54]:

```
fruits = {'Ali' : 'Apple', 'Chong' : 'Carrot', 'Gobi' : 'Orange', 'Alan' :␣
 ↪'Carrot', 'Vinston' : 'Coconut'}
print(fruits)

for name, fruit in fruits.copy().items():
   if fruit == 'Carrot':
      del fruits[name]
print(fruits)
```

```
{'Ali': 'Apple', 'Chong': 'Carrot', 'Gobi': 'Orange', 'Alan': 'Carrot',
'Vinston': 'Coconut'}
{'Ali': 'Apple', 'Gobi': 'Orange', 'Vinston': 'Coconut'}
```

1.2.2 Strategy: Create a new collection

[55]:

```
fruits = {'Ali' : 'Apple', 'Chong' : 'Carrot', 'Gobi' : 'Orange', 'Alan' :␣
 ↪'Carrot', 'Vinston' : 'Coconut'}
print(fruits)

fruits_not_carrot = {}
for name, fruit in fruits.copy().items():
   if fruit != 'Carrot':
      fruits_not_carrot[name] = fruit
print(fruits_not_carrot)
```

```
{'Ali': 'Apple', 'Chong': 'Carrot', 'Gobi': 'Orange', 'Alan': 'Carrot',
'Vinston': 'Coconut'}
{'Ali': 'Apple', 'Gobi': 'Orange', 'Vinston': 'Coconut'}
```

1.3 The range() Function

[56]:

```
for i in range(5):
   print(i)
for i in range(5):
   print(i, end=' ')
```

```
0
1
2
3
4
0 1 2 3 4
```

[57]:

```
for i in range(2, 10):    # does not include 10
    print(i, end=' ')
print()

for i in range(2, 11, 3): # does not include 11
    print(i, end=' ')
print()

for i in range(10, 2, -3):
    print(i, end=' ')
print()

for i in range(-10, 20, 4):
    print(i, end=' ')
print()

for i in range(-10, -20, -2):  # does not include -20
    print(i, end=' ')
print()
```

```
2 3 4 5 6 7 8 9
2 5 8
10 7 4
-10 -6 -2 2 6 10 14 18
-10 -12 -14 -16 -18
```

[58]:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
print(a)

length = len(a)
print(length)

for i in range(length):
    print(i, a[i])

for i in range(len(a)):
    print(i, a[i])
```

```
['Mary', 'had', 'a', 'little', 'lamb']
5
0 Mary
1 had
2 a
3 little
4 lamb
0 Mary
1 had
2 a
```

```
3 little
4 lamb
```

[118]:

```
# STRANGE !!!!
# A strange thing happens if you just print a range:
print(range(10))
```

```
range(0, 10)
```

> NOTE: In many ways the object returned by `range()` (https://docs.python.org/3/library/stdtypes.html#range) behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space. We say such an object is **iterable** (https://docs.python.org/3/glossary.html#term-iterable).

The function `sum()` and `list()` takes in an **iterable**.

[119]:

```
sum(range(1,5))   # sum 1 + 2 + 3 + 4
```

[119]:

```
10
```

[120]:

```
list(range(4))    # get a list from a range
```

[120]:

```
[0, 1, 2, 3]
```

1.4 `break` and `continue` Statements, and `else` Clauses on Loops

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the iterable (with `for`), i.e. when no `break` occurs or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement.

[121]:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement, also borrowed from C, continues with the next iteration of the loop:

[122]:
```
for num in range(2, 10):
   if num % 2 == 0:
      print("Found an even number", num)
      continue
   print("Found a number", num)
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

1.5 pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

[ ]:
```
while True:
   pass
```

[ ]:
```
class MyEmptyClass:
   pass
```

[ ]:
```
def initlog(*args):
   pass   # Remember to implement this!
```

1.6 Defining Functions

[1]:
```
def fib(n):    # write Fibonacci series up to n
   """Print a Fibonacci series up to n."""
   a, b = 0, 1
   while a < n:
      print(a, end=' ')
      a, b = b, a+b
   print()

# Now call the function we just defined:
fib(2000)

f = fib
f(2000)
```

```

print(fib)
print(f)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
<function fib at 0x7f1bc4312b80>
<function fib at 0x7f1bc4312b80>
```

the ””” line is optionally a string literal which is the function’s *Documentation String* (https://docs.python.org/3/tutorial/controlflow.html#tut-docstrings) or *docstring*.

Arguments are passed using call by value (where the value is always an object reference, not the value of the object).

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

[2]:

```
fib(0)
print(fib(0))
```

```
None
```

functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it’s a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`

[20]:

```
def fib2(n):  # return Fibonacci series up to n
   """Return a list containing the Fibonacci series up to n."""
   result = []
   a, b = 0, 1
   while a < n:
      result.append(a)    # see below
      a, b = b, a+b
   return result

print( fib2(100) )
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

1.7 More on Defining Functions

1.7.1 Default Argument Values

[6]:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)

ask_ok("yes or no ? :")
```

```
yes or no ? : y
```

[6]:

```
True
```

[10]:

```
ask_ok("yes or no ? :", 3)
```

```
yes or no ? : v

Please try again!

yes or no ? : ye
```

[10]:

```
True
```

[8]:

```
ask_ok("yes or no ? :", 2, "come one!")
```

```
yes or no ? : v

come one!

yes or no ? : v

come one!

yes or no ? : v
```

```
        ␣
↪---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call␣
↪last)
```

```
    <ipython-input-8-05e08a5aa795> in <module>
----> 1 ask_ok("yes or no ? :", 2, "come one!")


    <ipython-input-6-35c3e22d3e74> in ask_ok(prompt, retries, reminder)
      8         retries = retries - 1
      9         if retries < 0:
---> 10             raise ValueError('invalid user response')
     11         print(reminder)
     12 


    ValueError: invalid user response
```

[9]:

```python
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

```
5
```

The default values are evaluated at the point of function definition in the defining scope

[14]:

```python
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

```
[1]
[1, 2]
[1, 2, 3]
```

Important warning: The default value is **evaluated only once**. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

[17]:

```python
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

```
print(f(1))
print(f(2))
print(f(3))
```

```
[1]
[2]
[3]
```

the default NOT shared between subsequent calls,

1.7.2 Keyword Arguments

Keyword parameters are also referred to as named parameters.

[9]:

```
def fun(name, verb='likes', thing='apples', freq='everyday'):
   print(name, verb, thing, freq, '!!')

fun('Ali')                          # 1 positional argument
fun(name='Chong')                   # 1 keyword argument
fun(name='Alan', thing='oranges')   # 2 keyword arguments
fun(thing='oranges', name='Alan')   # 2 keyword arguments
fun('Gobi', 'loves', 'Coconuts')    # 3 positional arguments
fun('Gobi', freq='every month')     # 1 positional, 1 keyword argument
```

```
Ali likes apples everyday !!
Chong likes apples everyday !!
Alan likes oranges everyday !!
Alan likes oranges everyday !!
Gobi loves Coconuts everyday !!
Gobi likes apples every month !!
```

Variable Length Positional Arguments : formal parameter of the form `*name`, receives a tuple containing the positional arguments beyond the formal parameter list.
Variable Length Keyword Arguments formal parameter of the form `**name`, receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter. `*name` must occur before `**name`

[36]:

```
def cheeseshop(kind, *arguments, **keywords):
    print("arguments=", arguments)
    print("keywords=",keywords)
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
arguments= ("It's very runny, sir.", "It's really very, VERY runny, sir.")
keywords= {'shopkeeper': 'Michael Palin', 'client': 'John Cleese', 'sketch':
'Cheese Shop Sketch'}
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

> Note the output: - `arguments` contains `tuple` containing the positional arguments - `keywords` contains dictionary containing all keyword arguments

1.7.3 Special parameters

[1]:

```
import sys
print (sys.version)
print (sys.version_info)
```

```
3.8.1 (default, Jan  8 2020, 22:29:32)
[GCC 7.3.0]
sys.version_info(major=3, minor=8, micro=1, releaselevel='final', serial=0)
```

[33]:

```
# A function definition may look like:
#
# def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
#       -----------    ----------     ----------
#         |             |                  |
#         |        Positional or keyword   |
#         |                                - Keyword only
#          -- Positional only
#
# where / and * are optional.
```

> / and * works in Python version 3.8 and above.

[23]:

```
def standard_arg(arg):
    print(arg)
```

```
def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)

standard_arg(1)       # ok
standard_arg(arg=2)   # ok

pos_only_arg(3)       # ok

kwd_only_arg(arg=4)   # ok

combined_example(5, 6, kwd_only=7)              # ok
combined_example(8, standard=9, kwd_only=10)    # ok
```

```
1
2
3
4
5 6 7
8 9 10
```

[24]:

```
pos_only_arg(arg=11)   # NOT ok
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-24-44f8949cfcf2> in <module>
   ----> 1 pos_only_arg(arg=11)   # NOT ok


       TypeError: pos_only_arg() got some positional-only arguments passed as␣
↪keyword arguments: 'arg'
```

[25]:

```
kwd_only_arg(12)   # NOT ok
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-25-657a760e10a2> in <module>
   ----> 1 kwd_only_arg(12)  # NOT ok


       TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given
```

```
[26]: combined_example(8, standard=9, 10)   # NOT ok
```

```
         File "<ipython-input-26-9e8b02ee4952>", line 1
       combined_example(8, standard=9, 10)   # NOT ok
                                      ^
   SyntaxError: positional argument follows keyword argument
```

```
[68]: def foo(name, **kwds):
          return 'name' in kwds

      foo(1, **{'name': 2})   # will produce ERROR because got multiple values for␣
       ↪argument 'name' !!
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-68-8c980cdd63b2> in <module>
         2     return 'name' in kwds
         3
   ----> 4 foo(1, **{'name': 2})   # will produce ERROR !!


       TypeError: foo() got multiple values for argument 'name'
```

```
[5]: def foo(name, /, **kwds):  # ADDED / (positional only arguments)
         return 'name' in kwds
```

```
foo(1, **{'name': 2})    # now it is okay.
```

[5]:
```
True
```

As guidance:
- Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
- Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
- For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

1.7.4 Arbitrary Argument Lists

- specify that a function can be called with an arbitrary number of arguments.
- These arguments will be wrapped up in a tuple (Tuples and Sequences).
- Before the variable number of arguments, zero or more normal arguments may occur.
- ___Any formal parameters which occur after the *args parameter are ‘keyword-only’ arguments___

[106]:
```
def fun(separator, *args):    # args is a tuple containing arguments
    print("arguments passed inside function is: ", args)
    s = separator.join(args)
    return s

# words in a tuple
wordlist = ("apple", "orange", "coconut", "banana")
separator = '+'
s = separator.join(wordlist)
print(s)

# words in a list
wordlist = ["apple", "orange", "coconut", "banana"]
separator = '+'
s = separator.join(wordlist)
print(s)

s = fun('+', "apple", "orange", "coconut", "banana")
print(s)
```

```
apple+orange+coconut+banana
apple+orange+coconut+banana
arguments passed inside function is:  ('apple', 'orange', 'coconut', 'banana')
```

```
apple+orange+coconut+banana
```

[107]:

```
def fun(*args, separator):   # args is a tuple containing arguments
    print("arguments passed inside function is: ", args)
    s = separator.join(args)
    return s

# '+' can only be passed in as keyword argument
# s = fun("apple", "orange", "coconut", "banana", '+') # This will NOT work

s = fun("apple", "orange", "coconut", "banana", separator='.')
print(s)
```

```
arguments passed inside function is:  ('apple', 'orange', 'coconut', 'banana')
apple.orange.coconut.banana
```

1.7.5 Unpacking Argument Lists

When the arguments are already in a list or tuple but **need to be unpacked** for a function Solution: pass the list or tuple as argument `*list`

[108]:

```
def fun(*args, separator):   # args is a tuple containing arguments
    print("arguments passed inside function is: ", args)
    s = separator.join(args)
    return s

wordlist = ("apple", "orange", "coconut", "banana")  # wordlist already is a␣
 ↪tuple
s = fun(*wordlist, separator='.')                   # note the '*' in front of␣
 ↪wordlist
print(s)

wordlist = ["apple", "orange", "coconut", "banana"]   # wordlist already is a␣
 ↪list
s = fun(*wordlist, separator='.')                     # note the '*' in front of␣
 ↪wordlist
print(s)
```

```
arguments passed inside function is:  ('apple', 'orange', 'coconut', 'banana')
apple.orange.coconut.banana
arguments passed inside function is:  ('apple', 'orange', 'coconut', 'banana')
apple.orange.coconut.banana
```

[66]:

```
r = range(3, 6)
print(r)
print(list(r))
```

```
n = (3, 6)
r = range(*n)    # n already is a tuple, *n would unpack it
print(r)
print(list(r))
```

```
range(3, 6)
[3, 4, 5]
range(3, 6)
[3, 4, 5]
```

[104]:

```
def fun(a, b, c, d):
    s = a + '.' + b + '.' + c + '.' + d
    return s

wordlist = ("apple", "orange", "coconut", "banana")  # wordlist is a tuple
s = fun(*wordlist)                                   # note the '*' in front of wordlist
print(s)

wordlist = ["apple", "orange", "coconut", "banana"]  # wordlist is a list
s = fun(*wordlist)                                   # note the '*' in front of wordlist
print(s)
```

```
apple.orange.coconut.banana
apple.orange.coconut.banana
```

[105]:

```
def fun(name, id, title):
    return title + ' ' + name + '(' + id + ')'

s = fun(name='Chong', id='4123', title='Mr.')       # arguments directly using␣
 ↪keywords
print(s)

record = {'name':'Chong', 'id':'4123', 'title':'Mr.'}  # record is a dictionary
s = fun(**record)                                   # note the '**' in front of record
print(s)

record = {'id':'4123', 'title':'Mr.', 'name':'Chong'}  # record is a dictionary
s = fun(**record)                                   # note the '**' in front of record
print(s)
```

```
Mr. Chong(4123)
Mr. Chong(4123)
Mr. Chong(4123)
```

[110]:

```
def fun(**args):     # parameter now a variable keyword list
    print(args)
    return args['title'] + ' ' + args['name'] + '(' + args['id'] + ')'
```

```
s = fun(name='Chong', id='4123', title='Mr.')       # arguments directly using␣
 ↪keywords
print(s)

record = {'name':'Chong', 'id':'4123', 'title':'Mr.'}  # record is a dictionary
s = fun(**record)                          # note the '**' in front of record
print(s)

record = {'id':'4123', 'title':'Mr.', 'name':'Chong'}  # record is a dictionary
s = fun(**record)                          # note the '**' in front of record
print(s)
```

```
{'name': 'Chong', 'id': '4123', 'title': 'Mr.'}
Mr. Chong(4123)
{'name': 'Chong', 'id': '4123', 'title': 'Mr.'}
Mr. Chong(4123)
{'id': '4123', 'title': 'Mr.', 'name': 'Chong'}
Mr. Chong(4123)
```

1.7.6 Lambda Expressions

- Lambda functions can be used wherever function objects are required.
- syntactically restricted to a single expression.
- can reference variables from the containing scope.

[111]:

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)

print ( f(5) )
print ( f(3) )
print ( f(0) )
print ( f(8) )
```

```
47
45
42
50
```

[112]:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
print( pairs )

pairs.sort(key=lambda pair: pair[0])
print( pairs )
```

```
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

1.7.7 Documentation Strings

[113]:

```
def myfunc(a, b):
    """Just an ordinary function.

    This is just a simple function
    to test documentation strings
    of Python.

    Notice that the line after the title
    line is blank. Then indentation would
    follow the identation of the first line
    of this description.
    """
    return a+b

print(myfunc.__doc__)
```

```
Just an ordinary function.

    This is just a simple function
    to test documentation strings
    of Python.

    Notice that the line after the title
    line is blank. Then indentation would
    follow the identation of the first line
    of this description.
```

1.7.8 Function Annotations

- Function annotations are completely optional metadata information about the types used by user-defined functions (see PEP 3107 (https://www.python.org/dev/peps/pep-3107) and PEP 484 (https://www.python.org/dev/peps/pep-0484) for more information).
- Annotations are stored in the **annotations** attribute of the function as a dictionary
- have no effect on any other part of the function.
- Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation.
- Return annotations are defined by a literal ->, followed by an expression, between the parameter list and the colon denoting the end of the def statement.

- The following example has a positional argument, a keyword argument, and the return value annotated:

[114]:

```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

f('spam')
```

```
Annotations: {'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class
'str'>}
Arguments: spam eggs
```

[114]:

```
'spam and eggs'
```

1.8 Intermezzo: Coding Style

PEP 8 – Style Guide for Python Code
https://www.python.org/dev/peps/pep-0008/

- Use 4-space indentation, and no tabs.

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don't exceed 79 characters.

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

- When possible, put comments on a line of their own.

- Use docstrings.

- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.

- Name your classes and functions consistently; the convention is to use `UpperCamelCase` for classes and `lowercase_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see A First Look at Classes (https://docs.python.org/3/tutorial/classes.html#tut-firstclasses) for more on classes and methods).

- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.

- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

2 END OF The Python Tutorial -> More Control Flow Tools

[ ]: