California State University Fullerton

CPSC-223P

Python Programming

Anirudh Sahu

Python Tutorial
Section 4
Control Flow Tools

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print("Hello world!")
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Types of conditional Statements:

1. Different types of conditional statements in python :
   - If
   - If-else
   - Nested if
   - If-elif statements.
2. Comparisions:
   Equal: ==
   Not equal: !=
   Greater than: >
   Less than: <
   Greater than or equal to: >=
   Less than or equal to: <=
   Object identity: is (to test if two objects have same identity i.e if they are same object in the memory)
3. Boolean Operations:
   - and
   - or
   - not
4. 0/ ''/[]/{}/()/None: evaluates to false

   <u>example:</u>

   ```
   condition= None
       if condition:
           print("evaluated to true")
       else:
           print("evaluate to false")
   ```

Output: evaluated to false

4.1 `if` Statements

- Syntax
  - `if`
  - `elif`
  - `else`
- No parentheses as in other languages
- A colon (`:`) is used to mark the end of the condition
- The keyword `elif` is short of 'else if'
- There can be zero or more `elif` parts
- The `else` part is optional
- An `if-elif-elif-…-else` sequence is a substitute for the switch/case/default statements found in other languages

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

4.2 `for` Statements

- Syntax
  - `for`
  - `in`
- No parentheses as in other languages
- A colon (`:`) is used to mark the end of the condition
- Python's `for` statement iterates over the items of any sequence (a list or a string)
- This is different than other languages:
  - *Pascal:* iterates over an arithmetic progression of numbers
  - *C*: user defines both the iteration step and halting condition
- Recall that lists are mutable
- Care should be taken when modifying a list within a for loop

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

```
# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 The `range()` Function

- The built-in `range()` function generates arithmetic progressions
- This is useful to iterate over a sequence of numbers

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

- The given end point is never part of the generated sequence
- Slicing starts at a number other than zero
- Specific increment values are allowed (called the 'step')
- Negative increments are allowed

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

4.3 The `range()` Function (cont.)

- To iterate over the indices of a sequence, combine `range()` and `len()`

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

- The object returned by `range()` behaves like a list, but is not a list
- It is an 'iterable' object, suitable as a target for functions and constructs that expect successive items until the supply is exhausted
- The `for` statement is such a construct
- The `sum()` function takes an iterable

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

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

4.4 `break`, `continue`, and `else` on Loops

- The `break` statement breaks out of the innermost enclosing `for` or `while` loop

- Loops may have an `else` clause
  - Executes through exhaustion of a `for` loop
  - Executes when `while` loop condition is false
  - Does not execute when loop is terminated by a `break` statement

```
>>> 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
```

4.4 `break`, `continue`, and `else` on Loops (cont.)

- The `continue` statement continues with the next iteration of the loop

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 `pass` Statements

- The `pass` statement does nothing
- Can be used when a statement is required by syntax but the program requires no action

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

- Commonly used for creating minimal classes

```
>>> class MyEmptyClass:
...     pass
...
```

- Can be used as a place-holder for a function or conditional body during development
- Develop at abstract level, then come back and build in context later

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6 Defining Functions

- The keyword `def` introduces a function definition
- It must be followed by the function name and the parenthesized list of formal parameters
- A colon (`:`) is used to mark the end of the function header
- The statements that form the body of the function start at the next line, and must be indented
- The first statement of the function body can optionally be a string literal
  - This string literal is the function's documentation string, or 'docstring'
  - There are tools which use docstrings to automatically produce online or printed documentation

```
>>> 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)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

4.6 Defining Functions (cont.)

- The execution of a function introduces a new symbol table used for the local variables of the function
- Variable references look:
  - in the local symbol table
  - then in the local symbol tables of enclosing functions
  - then in the global symbol table
  - and finally in the table of built-in names.
- The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called
- Arguments are passed using call by value
  - the value is always an object reference, not the value of the object

```
>>> 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)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

4.6 Defining Functions (cont.)

- The statement `result.append(a)` calls a method of the list object result
- A method is a function that 'belongs' to an object and is named obj.methodname
  - obj is some object (this may be an expression)
  - methodname is the name of a method that is defined by the object's type
- The `return` statement returns with a value from a function

```
>>> 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
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

- Functions without a `return` statement do return the value `None`
- Use `print()` to see the value

```
>>> fib(0)
>>> print(fib(0))
None
```

4.7.1 Functions – Default Argument Values

- This creates a function that can be called with fewer arguments than it is defined to allow
- This function can be called in several ways:
  - giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
  - giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
  - or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`
- The `in` keyword tests whether or not a sequence contains a certain value
- The default values are evaluated at the point of function definition in the defining scope
- This will print 5

```
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)
```

```
i = 5

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

i = 6
f()
```

4.7.1 Functions – Default Argument Values (cont.)

- The default value is only evaluated once
- Consider mutable objects such as lists, dictionaries, or class instances

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

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

- This will print

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

- If you don't want the default to be shared between subsequent calls, you can write the function like this instead

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

4.7.2 Functions – Keyword Arguments

- Functions can be called using keyword arguments of the form `kwarg=value`
- The example function accepts
  - one required argument (`voltage`)
  - three optional arguments (`state, action, and type`)
- The example function can be called
  - 1 positional argument
  - 1 keyword argument
  - 2 keyword arguments
  - 2 keyword arguments (order not required)
  - 3 positional arguments
  - 1 positional argument, 1 keyword argument
- Invalid function calls
  - Required argument missing
  - Non-keyword argument after keyword argument
  - Duplicate value for the same argument
  - Unknown keyword argument

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='VOOOOOM')
parrot(action='VOOOOOM', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

```
parrot()
parrot(voltage=5.0, 'dead')
parrot(110, voltage=220)
parrot(actor='John Cleese')
```

4.7.2 Functions – Keyword Arguments (cont.)

- When a final formal parameter of the form `**name` is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter
- This may be combined with a formal parameter of the form `*name` which receives a tuple containing the positional arguments beyond the formal parameter list
- `*name` must occur before `**name`
- The order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call

```
def cheeseshop(kind, *arguments, **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")
```

```
-- 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
```

4.7.3 Functions – Special Parameters

- By default, arguments may be passed to a Python function either by position or explicitly by keyword

- For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by
  - position
  - position or keyword
  - Keyword

4.7.3 Functions – Special Parameters (cont.)

- If / and * are not present in the function definition, arguments may be passed to a function by position or by keyword
- Positional-only parameters are placed before a / in the argument list
- Keyword-only parameters are placed after a * in the argument list

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

- Most familiar form places no restrictions on the calling convention and arguments may be passed by position or keyword

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

4.7.3 Functions – Special Parameters (cont.)

- The position-only form is restricted to only use positional parameters as there is a `/` in the function definition

```
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got an unexpected keyword argument 'arg'
```

- The keyword-only form is restricted to only use keyword arguments as there is a `*` in the function definition

```
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was
given

>>> kwd_only_arg(arg=3)
3
```

4.7.3 Functions – Special Parameters (cont.)

- Example of all three calling conventions
  - position-only
  - position-or-keyword
  - keyword-only

- Use positional-only if you want the name of the parameters to not be available to the user and they have no real meaning
- Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names

```
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
...
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3
were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got an unexpected keyword argument
'pos_only'
```

4.7.4 Functions – Arbitrary Argument Lists

- A variable number arguments is specified by the form `*name` and is passed as a tuple
  - A tuple consists of a number of values separated by commas

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

- Before the variable number of arguments, zero or more normal arguments may occur

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

- Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments

4.7.5 Functions – Unpacking Argument Lists

- The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments
- For instance, the built-in `range()` function expects separate start and stop arguments
- If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple

```
>>> list(range(3, 6))    # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))   # call with arguments unpacked from a list
[3, 4, 5]
```

- In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised",
"action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through
it. E's bleedin' demised !
```

4.7.6 Functions – Lambda Expressions

- Small anonymous functions can be created with the `lambda` keyword
- Lambda functions can be used wherever function objects are required
- They are syntactically restricted to a single expression
- Like nested function definitions, lambda functions can reference variables from the containing scope
- Another use is to pass a small function as an argument

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

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

4.7.7 Functions – Documentation Strings

- The first line should always be a short, concise summary of the object's purpose
  - This line should begin with a capital letter and end with a period
- If there are more lines in the documentation string, the second line should be blank
- The remaining lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.7.8 Functions – Function Annotations

- Function annotations are completely optional metadata information about the types used by user-defined functions
- Annotations are stored in the `__annotations__` attribute of the function as a dictionary and 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

```
>>> 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'>, 'return': <class 'str'>,
'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.8 Coding Style

- PEP 8 (Python Enhancement Proposals) has emerged as the most used style guide
- PEP 8 promotes a very readable and eye-pleasing coding style
- PEP 8 most import points:
  - 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.
- PEP 8 most import points (cont.):
  - 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.
  - 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.