

COMP(2041|9044) 26T2 — Python Functions

<https://www.cse.unsw.edu.au/~cs2041/26T2/>

<https://www.cse.unsw.edu.au/~cs2041/26T2/>

COMP(2041|9044) 26T2 — Python Functions

1 / 30

Programming Paradigms

A programming paradigm is a **style of programming** that defines how programs are designed and structured.

- Procedural programming
 - Breaks a large problem into smaller procedures (functions)
 - Examples: C, Pascal, Fortran
- Object-Oriented Programming (OOP)
 - Organizes programs around objects, rather than just functions
 - Examples: C++, Java, C#
- Functional programming
 - Treats computation as the evaluation of functions
 - Examples: Haskell, Lisp
- Declarative programming
 - Describes what is wanted rather than how to compute it
 - Examples: SQL, HTML, CSS
- Logic programming
 - Programs consist of facts and rules. Ask questions instead of writing algorithms
 - Examples: Prolog, Datalog

<https://www.cse.unsw.edu.au/~cs2041/26T2/>

COMP(2041|9044) 26T2 — Python Functions

2 / 30

Multi-paradigm Support

- Many modern languages support multiple programming paradigms
- Python supports:
 - Procedural programming
 - Object-orient programming
 - Functional programming
 - Declarative programming (limited)

<https://www.cse.unsw.edu.au/~cs2041/26T2/>

COMP(2041|9044) 26T2 — Python Functions

3 / 30

Defining Python Functions

- Python functions can be defined, like C, with a fixed number of parameters

```
def polly(x, a, b, c):  
    return a * x ** 2 + b * x + c
```

- functions can be called, like C, with **positional** arguments

```
>>> polly(3, 5, -3, 6)
```

- or with **keyword** arguments

```
>>> polly(a=5, c=6, b=-3, x=3)
```

Or with both **positional** and **keyword** arguments (**keyword must follow positional**)

```
>>> polly(3, c=6, b=-3, a=5)
```

- functions can restrict how they are called using special parameters / and *
 - Parameters before / can only be passed by position
 - Parameters after * must be passed by keyword

Default Values for Function Arguments

- default values can be specified for function parameters, but **non-default parameters must come first**

```
def polly(x, a=1, b=2, c=0):  
    return a * x ** 2 + b * x + c
```

- allowing functions to be called without specifying all parameters

```
>>> polly(3)  
15  
>>> polly(b=1, x=1)  
2
```

- convenient consequence - you can add an extra parameter to a function, without changing existing calls, by giving the parameter a default value

Mutable Default Parameter values are Dangerous

- the default value is a single instance created once when the function is defined, not each time it is called
- safe for immutable types: numbers, strings, ...
- unexpected results from mutable types: lists, dicts, ...
 - common bug in Python programs
 - can be used deliberately

```
def append_one(x = []):  
    x.append(1)  
    return x
```

```
>>> append_one()  
[1]  
>>> append_one()  
[1, 1]  
>>> append_one()  
[1, 1, 1]
```

```
def append_one(x = None):  
    if x is None:  
        x = []  
    x.append(1)  
    return x
```

```
>>> append_one()  
[1]  
>>> append_one()  
[1]  
>>> append_one()  
[1]
```

- The standard workaround:
 - Use None as the default value
 - Create the mutable object inside the function when needed

Variable Numbers of Function Arguments

- packing/unpacking operators `*` and `**` allow variable number of arguments.
 - Use `*` to pack positional arguments into **tuple**
 - Use `**` to pack keyword arguments into **dict**

```
def f(*args, **kwargs):  
    print('positional arguments:', args)  
    print('keywords arguments:', kwargs)
```

```
>>> f("COMP", 2041, 9044, answer=42, option=False)  
positional arguments: ('COMP', 2041, 9044)  
keywords arguments: {'answer': 42, 'option': False}
```

Unpacking Function Arguments

- `*` and `**` can be used in reverse for function calls
 - Use `*` to unpack iterable (e.g. list or tuple) into positional arguments
 - Use `**` to unpack dict into keyword arguments

```
>>> arguments = ['Hello', 'there', 'Andrew']  
>>> keyword_arguments = {'end': '!!!\n', 'sep': ' --- '}  
>>> print(arguments, keyword_arguments)  
['Hello', 'there', 'Andrew'] {'end': '!!!\n', 'sep': ' --- '}  
>>> print(*arguments, **keyword_arguments)  
Hello --- there --- Andrew!!!
```

No main function

- Python has no special “main” function called to started execution (unlike e.g C)
- importing a file executes any code in it
- special global variable `__name__` set to module name during import
- if a file is executed rather than imported, `__name__` set to special value `__main__`
- the common pattern

```
def main():  
    #body of the main function  
  
if __name__ == '__main__':  
    main()
```

- “if `__name__ == '__main__':`” means only run this code if this file is the starting point of the program

docstrings

- A Python Docstring is a string used to document a module or a function
 - Explain the purpose of code
 - Help other programmers understand how to use it
 - Allow Python tools (such as `help()`) to automatically generate documentation.
- Use triple-quotes `"""..."""` (preferred) or `'...'...`
- Placed immediately after the definition

```
def polly(x, a, b, c):  
    """calculate quadratic polynomial"""  
    return a * x ** 2 + b * x + c
```

- Every function and module has a special `__doc__` attribute.

```
>>> polly.__doc__  
'calculate quadratic polynomial'
```

```
>>> help(polly)  
Help on function polly in module __main__:  
  
polly(x, a, b, c)  
    calculate quadratic polynomial
```

docstrings

- for detailed documentation, use multiple lines

```
def polly(x, a, b, c):  
    """calculate quadratic polynomial  
    a -- squared component  
    b -- linear component  
    c -- offset  
    """  
    return a * x ** 2 + b * x + c
```

- docstring vs. comment
 - docstrings are used to explain what the code does, provided for users of the module and function
 - comments are used to explain how the code works, provided for programmers to read the code

- a variable assigned a value in a function is by default **local** to the function
- a variable not assigned a value in a function is by default **global** to entire program
- keyword **global** can be used to make variable global

local versus global variables

```
>>> x = 12
>>> def f():
...     x = 34
...     print(x)
>>> def g():
...     global x
...     x = 56
...     print(x)
>>> f()
34
>>> print(x)
12
>>> g()
56
>>> print(x)
56
```

variable scope - more complex example

```
def a():
    x = 1
    print('a', x, y, z)
def b():
    x = 2
    y = 2
    a()
    print('b', x, y, z)
def c():
    x = 3
    y = 3
    global z
    z = 3
    b()
    print('c', x, y, z)
```

```
>>> x = 4
>>> y = 4
>>> z = 4
>>> c()
a 1 4 3
b 2 2 3
c 3 3 3
```

[source code for scope.py](#)

- List comprehensions can be used to create lists (iterables) concisely.
- In simple cases, they are more readable than for loops or higher-order functions.
- They can be written as: **expression for value in iterable**

```
>>> [x**3 for x in range(10)]
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> [str(round(math.pi, digits)) for digits in range(1,7)]
['3.1', '3.14', '3.142', '3.1416', '3.14159', '3.141593']
```

- They can be written as: **expression for value in iterable if expression2**

```
>>> [x**3 for x in range(10) if x % 2 == 1]
[1, 27, 125, 343, 729]
```

- list comprehensions can be nested but this may less readable than use of loops

list comprehension example

```
def is_odd(number):
    return number % 2 != 2
def odd0(numbers):
    """extract odd_numbers from list using for loop"""
    odd_numbers = []
    for n in numbers:
        if is_odd(n):
            odd_numbers.append(n)
    return odd_numbers
def odd1(numbers):
    """extract odd_numbers from list using list comprehension"""
    return [n for n in numbers if is_odd(n)]
```

[source code for odd_numbers.py](#)

lambda - create a small anonymous function

- The keyword **lambda** provides creation of small anonymous functions
- **lambda** is useful for **higher-order programming** - passing functions to other functions.
- **lambda** allows the creation of a function within an expression.

```
>>> f = lambda x: x + 42
>>> type(f)
<class 'function'>
>>> f(12)
54
```

- **lambda** function body must be a single expression
 - function body can not contain statements such as while, return
 - better to define a named function if body is complex

lambda - variable binding

Beware variables in the lambda expression are bound when the lambda is evaluated, not when it is created.

```
>>> answer = 42
>>> f = lambda x: x + answer
>>> answer = 15
>>> f(12)
27
>>> answer = 34
>>> f(13)
47
```

Ugly workaround: make the variable the default value of a keyword argument.

```
>>> answer = 42
>>> f = lambda x, y=answer: x + y
>>> answer = 34
>>> f(12)
54
```

enumerate - builtin function

enumerate is used to iterate over an iterable while keeping track of the index of each element.

```
>>> languages = ['C', 'Python', 'Shell', 'Rust']
>>> list(enumerate(languages))
[(0, 'C'), (1, 'Python'), (2, 'Shell'), (3, 'Rust')]
>>> list(enumerate(languages, start=42))
[(42, 'C'), (43, 'Python'), (44, 'Shell'), (45, 'Rust')]
```

```
def my_enumerate(sequence, start=0):
    """return a list equivalent to the iterator returned
    by builtin function enumerate
    """
    n = start
    tuples = []
    for element in sequence:
        t = (n, element)
        tuples.append(t)
        n += 1
    return tuples
```

[source code for builtin.py](#)

zip - builtin function

zip is used to combine two or more iterables by pairing their corresponding elements

```
>>> languages = ['C', 'Python', 'Shell', 'Rust']
>>> editors = ['vi', 'emacs', 'atom', 'VScode', 'nano']
>>> list(zip(editors, languages))
[('vi', 'C'), ('emacs', 'Python'), ('atom', 'Shell'), ('VScode', 'Rust')]
```

```
def my_zip2(sequence1, sequence2):
    """return a list equivalent to the iterator returned by
    builtin function zip called with 2 sequences.
    Note: zip can be given any number of sequences."""
    tuples = []
    for index in range(min(len(sequence1), len(sequence2))):
        t = (sequence1[index], sequence2[index])
        tuples.append(t)
    return tuples
```

[source code for builtin.py](#)

list comprehension + zip example

```
def dot_product0(a, b):
    """return dot product of 2 lists - using for loop + indexing"""
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    return total
```

[source code for dot_product.py](#)

```
def dot_product2(a, b):
    """return dot product of 2 lists - using for loop + zip"""
    total = 0
    for x, y in zip(a, b):
        total += x * y
    return total
```

[source code for dot_product.py](#)

map - builtin function

map applies a function to every item in an iterable (such as a list, tuple, or string) and returns a map object (an iterator) containing the results.

```
>>> list(map(str, range(10)))
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> list(map(lambda x: x**3, range(10)))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> list(map(lambda x, y: x**y, range(10), range(10)))
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]
```

```
def my_map1(function, sequence):
    """return a list equivalent to the iterator returned by
    builtin function map called with 1 sequence.
    Note: map can be given more than 1 sequences."""
    results = []
    for value in sequence:
        result = function(value)
        results.append(result)
    return results
```

[source code for builtin.py](#)

list comprehension + map example

```
def multiply(x, y):
    """multiply 2 numbers - operator.mul does this"""
    return x * y
def dot_product4(a, b):
    """return dot product of 2 lists - map"""
    return sum(map(multiply, a, b))
def dot_product5(a, b):
    """return dot product of 2 lists - map + lambda"""
    return sum(map(lambda x, y: x * y, a, b))
def dot_product6(a, b):
    """return dot product of 2 lists - map + operator.mul"""
    return sum(map(operator.mul, a, b))
```

[source code for dot_product.py](#)

filter returns the elements of an iterable(s) such as list for which the supplied function returns true.

```
>>> list(filter(lambda x: x % 2 == 0, range(10)))  
[0, 2, 4, 6, 8]
```

```
def my_filter(function, sequence):  
    """return a list equivalent to the iterator returned by  
    builtin function filter called with a function.  
    Note: filter can be given None instead of a function."""  
    filtered = []  
    for value in sequence:  
        if function(value):  
            filtered.append(value)  
    return filtered
```

[source code for builtin.py](#)

filter + lambda example

```
def is_odd(number):  
    return number % 2 != 2
```

[source code for odd_numbers.py](#)

```
def odd2(numbers):  
    """extract odd_numbers from list using filter"""  
    return filter(is_odd, numbers)  
def odd3(numbers):  
    """extract odd numbers from list using filter + lambda"""  
    return filter(lambda n: n % 2 == 2, numbers)
```

[source code for odd_numbers.py](#)

sorted + lambda example

```
DAY_LIST = "Sunday Monday Tuesday Wednesday Thursday Friday Saturday".split()  
DAY_NUMBER = dict((day, number) for number, day in enumerate(DAY_LIST))  
def random_day_of_week():  
    return random.choice(DAY_LIST)  
def sort_days0(day_list):  
    return sorted(day_list, key=lambda day: DAY_NUMBER[day])  
def sort_days1(day_list):  
    return sorted(day_list, key=DAY_NUMBER.get)
```

[source code for sort_days.py](#)

The **functools** module provides more functions for higher-order programming, e.g.

```
>>> # sum first 10 positive integers
>>> functools.reduce(operator.add, range(1, 10))
45
>>> # multiply first 10 positive integers
>>> functools.reduce(operator.mul, range(1, 10))
362880
```

The **itertools** module provides functions for combining and constructing **iterators** allowing efficient handling of arbitrarily long sequences.

Type hints

- Python doesn't enforce types even when they are given, thus they are hints
- Static type checkers are common that do enforce types as much as possible
- For best results type enforcement should be including in your code
- Type hints help you and others read your code and are highly recommended

```
from typing import Optional, Union
```

```
a = 5
b = "Hello World"
# a type hint
c: int = 6
# but not enforced
d: int = "this isn't an int"
# composition of types
e: list[int] = [1, 2, 3, 4, 5]
# more composition of types
f: dict[int, list[tuple[str, str]]] = {1: [('a', 'b'), ('a', 'c')], 3: [('c',
↪ 's'), ('c', 'g')]}
```

Type hints

```
from typing import Optional, Union

# `Optional` allows for None values
g: Optional[float] = None
# `Union` allows for two or more types
h: Union[int, float] = 4
# type hints can also be used on function arguments and return values
def func(a: int, b: str = 'Hi\n') -> int:
    return len(b * a)
# for variables used in loops, tuple unpacking, or assignment can be
↪ pre-typed
# pre-typing does not define the variable as it has not assigned a value and
↪ python variables must always be initialised
j: int
for j in range(0, 100):
    pass

k: bool
if k := validate(data):
    pass

l: bool
m: int
n: str
l, m, n = (True, 99, "Apple")

# a variable's type can be changed by first deleting it then redefining it
o: int = 0
del o
o: str = ""
```