

# COMP(2041|9044) 22T2 — Python Functions

<https://www.cse.unsw.edu.au/~cs2041/22T2/>

# 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)  
42
```

- or with **keyword** arguments

```
>>> polly(a=5, c=6, b=-3, x=3)  
42
```

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

```
>>> polly(3, c=6, b=-3, a=5)  
42
```

- functions can restrict how they are called using special argument / and \*

# Default values for Functions Arguments

- default values can be specified for parameters

```
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
```

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

# Mutable Default values are dangerous

- the default value is a single instance
- fine for immutable types: numbers, strings, ...
- unexpected results from mutable types: lists, dicts, ...

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

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

# Mutable Default values - workaround

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

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

# Mutable Default values - workaround

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

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

# 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}
```

# Packing 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__`**
- so can call a function when a file is executed like this

```
if __name__ == '__main__':  
    initial_function()
```

“

# docstrings

- A Python Docstring is a string specified as first statement of function
- use `"""` triple-quotes

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

- provides documentation to human readers but also available for automated tools

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

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

- 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

## variable scope - 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)
```

source code for scope.py

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

# List Comprehensions

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

# 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 - 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** returns tuples pairing a count with members of an iterable such as a list.

```
>>> 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** returns tuples formed from corresponding members of iterables such as lists.

```
>>> 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

# 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

# map - builtin function

**map** calls a function with argument(s) taken from iterable(s) such as list(s) and returns the functions return values

```
>>> 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 + zip 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 - builtin function

**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

# exploring function for combining and constructing further

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.