Fundamentals of Computer Programming

Lecture slides - Functions

Intro

- This lesson covers
    - Importing and Using Functions
    - Defining Functions
    - Default and Keyword Arguments
    - Argument Lists
    - Lambda Expressions

Using externally defined functions

Part of Python language itself

- Up until now we have only used a couple of *built-in functions*, such as `print()`, `input()`, `range()` and `type()`

Depends on version of Python

- There are approximately 68 *built-in functions* available in Python
- However, many thousands of other functions exist, many of which are defined within the **Python Standard Library** Comes with most installations of Python
- Functions which are not *built-in* must be **imported** before they can be used within our programs. Functions are defined in **modules**
- We should only import what we need to use for a particular problem (reducing *namespace* clutter)

Using externally defined functions

- We can import a whole **module**, that contains many functions, e.g.

```
import math        # import the whole math module
```

- To refer to the imported functions we prefix the name with the module name. In this case “`math.`” e.g.

```
print("Hypot. of the triangle is", math.sqrt(a * a + b * b))
```

- Importing a module also allows access to constant values, e.g.

```
print("Area of the circle is", (2 * math.pi * radius))
```

↳ Already defined as part of the module

Importing specific functions

- We can import a specific function if we like, e.g. Only allows you to use that part of the module

```
from math import sqrt      # import the sqrt() function directly
```

- When imported directly we no longer need to use the module prefix, e.g.

```
print("Hypot. of the triangle is", sqrt(a * a + b * b))
```

- We can directly import all module content using a wildcard (*), although this is not recommended since it *pollutes* the *namespace*, e.g.

```
from math import *

# all math functions and constants now available, without prefix
```

Can cause problems and accidental overwrites or logic errors

Importing “Types” from modules

- As we have seen, we can import predefined functions and constant values from modules
- We can also import predefined *data-types*
- We have already used the *built-in* data-types (`int`, `float`, `str`, etc.)
- However more complex data-types are often required, and like *functions* these can be accessed via *modules*
- An imported *data-type* not only has a specific name, but often a number of functions specifically related to that type

Example: The 'Decimal' type

Anything that has a decimal place (number)

- The `float` type is not always suitable for handling monetary applications
- Floating point numbers are stored as an *approximation* of a value with a fractional part, rather than been specifically designed to perform arithmetic like humans
- Hence subtle errors within calculations may occur, e.g. `print(1.1 + 2.2)`

```
3.3000000000000003
```

- The *Python Standard Library* contains a *decimal* module, which includes a `Decimal` type for use with monetary applications
- Decimals can perform arithmetic with integers, but not with floating-point numbers

Using the 'Decimal' type

- The Decimal type is NOT built-in to the language ↗ Part of the Python Standard library
- Hence, it needs to be imported before use, using the **from…import** statement, e.g.

```
from decimal import Decimal

# later in the same program….
print(Decimal("1.1") + Decimal("2.2"))
3.3
```

- Since Python relies so much on libraries, one or more **import** statements appear in almost all non-trivial programs

Defining Functions

- Rather than just rely on the *built-in* functions or those we can *import*, we can create our own

- We *define* our own functions using the **def** keyword

- A function definition includes a *name* and *formal parameters* (which identify what argument values must be passed during a call)
  Saying what you expect to be passed

- The statements associated with the function (i.e. the code) are provided as an indented block (in the same way as `if`, `for` and `while` statements)

- The first statement of the function body is usually a **docstring**
  documents what the function does
  - small concise + clear

Documentation Strings

- A **docstring** is a triple quoted sentence that appears as the first line of a function
- They should be a short, concise description of the function's purpose
- External tools use *docstrings* to automatically produce online or printed documentation
- The text should begin with a capital letter and end with a period (.)
- If multiple lines exist, the second line should be blank, to separate the heading from the rest of the description
- It's good practice to include *docstrings* in code that you write, so do this!

A very simple function

- Define a simple function that takes no *formal parameters* (notice the ':')

```
def displayTitle():
    """Displays the title of a movie."""
    print("Life of Brian")
```

- Once a function has been defined, we can *call* it (multiple times if we wish) somewhere else in our code using the *name* followed by *parentheses*, e.g.

```
displayTitle()

displayTitle()

displayTitle()
```

A function with parameters

↗ Names don't matter - as long as they don't clash with other variables

Essentially a list of variables separated by commas

```
def findMax(a,b):
    """Finds the maximum of two values."""
    if ( a > b ):
        max = a
    else:
        max = b
    return max
```

- When calling a function defined with formal parameters, we need to specify (pass) the actual parameter values to be used, e.g.

```
print("The maximum of the numbers you entered was",  findMax(num1, num2))
```

can be anything - values for A & B

- In this example the values stored in `num1` and `num2` are the *actual parameters* (arguments) to be passed to the function

More about functions

- The *actual parameters* (arguments) get passed to the function, and become accessible within the function block using the *formal parameters*.
- The *formal parameter* names act like **local variables** within the function

  only exist within the function

- Variables defined in the function block exist ONLY within that function, and cease to exist once the function ends (unless prefixed by `global`)
- A function can return a value using the **return** statement.
- If a **return** statement is not provided, the function will return the value **`None`**

  classed as a type of value even though it is nothing

Default arguments

Gives the parts of a function a default value if it has not already been specified.

- *Default arguments* allow a function to be called without providing ALL of the specified parameters
- These are specified using a '=' symbol in the function header, e.g.

```
def shouldContinue(prompt, answer=False):
    # function body
```

Default values can only be specified to the right of parameters without default values.

- Any parameter that is not provided uses the specified default value, e.g.

```
answer = shouldContinue("Do you wish to continue?")
```

- Default arguments can only be specified to the right of parameters that do not have defaults provided

Keyword arguments

Where name of parameter can be used when making the function call.

- Functions can be called using *keyword arguments*, where the name of the parameter is passed within the function call
- In most cases, keyword argument names must match an argument accepted by the function (see later for use of '**')
- The order of the passed keyword arguments does NOT need to match the formal parameter list
- But any provided keyword arguments must follow positional arguments
- No argument may receive a value more than once
- Any missing keyword argument uses the default argument value

Keyword arguments - examples

Has default values

```
def showMsg(title, body="", prefix="INFO", suffix="."):

    print(prefix,title,body,suffix)
```

- The above function could be called as follows -

```
showMsg("File opened")
```

values without a default value will fill with this.

calls show message

prefix defined as 'ERROR'

```
showMsg("File not opened", prefix="ERROR" )
```

title

Keyword argument

Instead of just passing a string it has the name of an argument within it. This means there is a relationship between the names.

Allows specification of what is passed where

Anything not set will go to its default value

```
showMsg("File missing", body="Insert Disk", suffix="Press return" )


showMsg(body="Calculation complete")  # this is not allowed, no 'title'
```

Variable Length Argument Lists

- As we already know from using the built-in `print()` function, some functions can take a **variable** number of arguments, e.g.

```
print("Hello this has 1 argument")
print("Hello", "there", "this has", 6, "arguments")
```

multiple arguments passed is called a Varadic list.

- When we define our functions, we can specify the same type of behaviour, i.e. we can define a function that takes a variable number of arguments

Similar to lists - covered later

- We achieve this by using **Tuples**, which are an important compound type in Python that we will discuss in future lessons
- Such *variadic* arguments are normally defined last in the formal parameter list (and can only be followed by keyword type parameters)

Variable Arguments Example

- As an example, let's define a function that takes an arbitrary number of strings and returns a file path

- Notice the use of the '`*`' before the parameter '`names`', this indicates that the number of passed argument values can be variable in length

* shows that it can take multiple values

```
def make_path(*names):
    result = ""
    for name in names:
        result += name + "/"
    return result
```

- In many ways a **Tuple** is similar to a list, which means we can access each of the passed arguments using a `for..in` style loop

Variable Arguments Example

- Once the function has been defined, calls can be made as usual
- However the number of actual parameters passed can vary, e.g.

```
# call the function with 2 arguments
print(make_path("home", "docs"))
```

```
home/docs/
```

```
# call the same function, but with 4 arguments
print(make_path("home", "code", "python", "sorter"))
```

```
home/code/python/sorter/
```

Arbitrary Keyword Arguments

Caller can pass any arguments they like - do not have to be defined within the function itself

- Finally, it is possible to define a function that accepts *arbitrary keyword arguments*, i.e. keyword arguments not explicitly named as parameters
- If a parameter name is prefixed with two '**' then any keyword argument passed which is *not* named gets received by that parameter

Not a tuple like other

- We process such arguments as a **Dictionary**, which will be discussed in future lessons
- For now, just recognise that a parameter name prefixed with two asterisks is valid
- A maximum of one parameter of this type can be provided, and it must appear after any *variable length argument* (identified with '*')

Arbitrary Keyword Example

- As an example, let's define a function that takes a *keyword argument* '`title`' and an *arbitrary keyword argument* '`info`' -

```
def show_details(title="Details", **info):
    print(title)
    for name in info:
        print(name, ":", info[name])
```

- Notice the use of the '`**`' before the parameter '`info`', this indicates that this should receive any unknown keyword parameters
- The above function could be called as follows -

```
show_details(title="warning")
show_details(msg="file created", err="no issues")
show_details(title="error", reason="disk full")
```

- Any passed parameter other than '`title`' will be received by '`info`'

Lambda Expressions

An anonymous function - when defined it has no name

- It is possible to define small 'anonymous' functions (a function that has no name), these are often implemented using **Lambda** expressions
- This may seem obscure at first, but it allows us to use functions as regular values meaning we can store them in variables, pass them as arguments, or even return them from other functions
- Lambda expressions are defined using the `lambda` keyword, followed by formal parameters, and a single expression, e.g. define a simple Lambda expression that squares the given value -

```
sqr = lambda num: num * num
```

All on one line - doesn't need a code block

```
# now use the function
print("10 squared is", sqr(10))
```

Lambda Expression

- Lambda expressions are limited to a single expression (no code block)
- The example below defines a function which when called returns a lambda expression, which itself can then be called -

```
def make_multi(m):
    """ returns a lambda expression, that multiplies by given value
"""
    return lambda num: num * m

f = make_multi(10)    # this returns a lambda expression (stored in f)
f(5)                  # this calls the returned function
50
```

- Although it may not seem so now, lambda expressions are very useful in many situations. For now just be aware they exist.

Example Lambda Expression

- Built-in functions, such as `sorted()`, allow lambda expressions to be passed as parameters, e.g. given a list (of lists) such as -

```
students = [ ["mark", 50], ["john", 20], ["mike", 25] ]
```

Built in sorting function

```
sorted(students)            # sorts using natural order
[['john', 20], ['mark', 50], ['mike', 25]]
```

By default will sort by first value.

- The *keyword-argument* 'key' specifies a function of one argument that is used to extract a comparison *key* from each element, e.g.

Expects a function

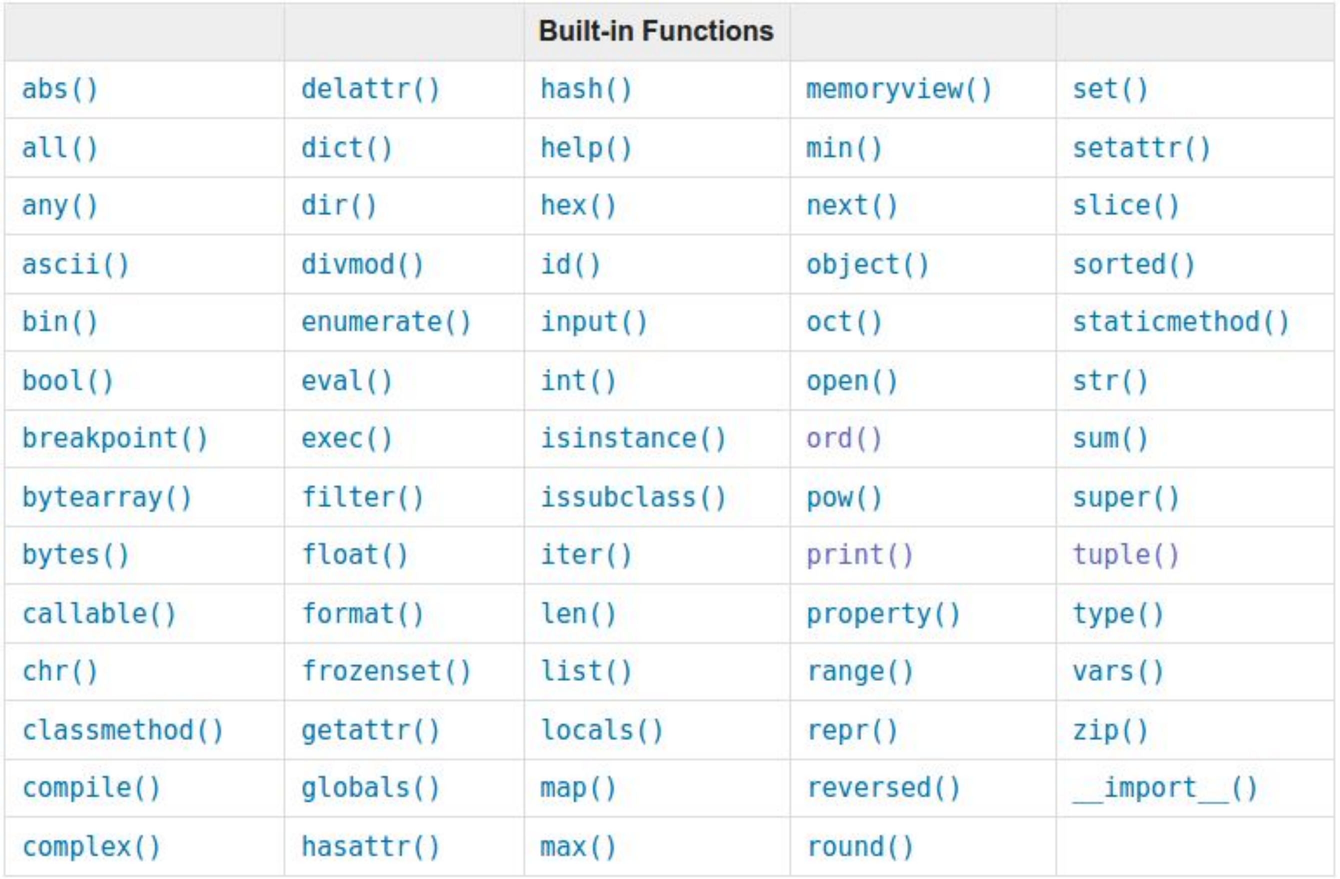
```
sorted(students, key=lambda s: s[1])   # compare using age instead
[['john', 20], ['mike', 25], ['mark', 50]]
```

- In this example, a lambda expression is being passed as an argument to another function

Summary

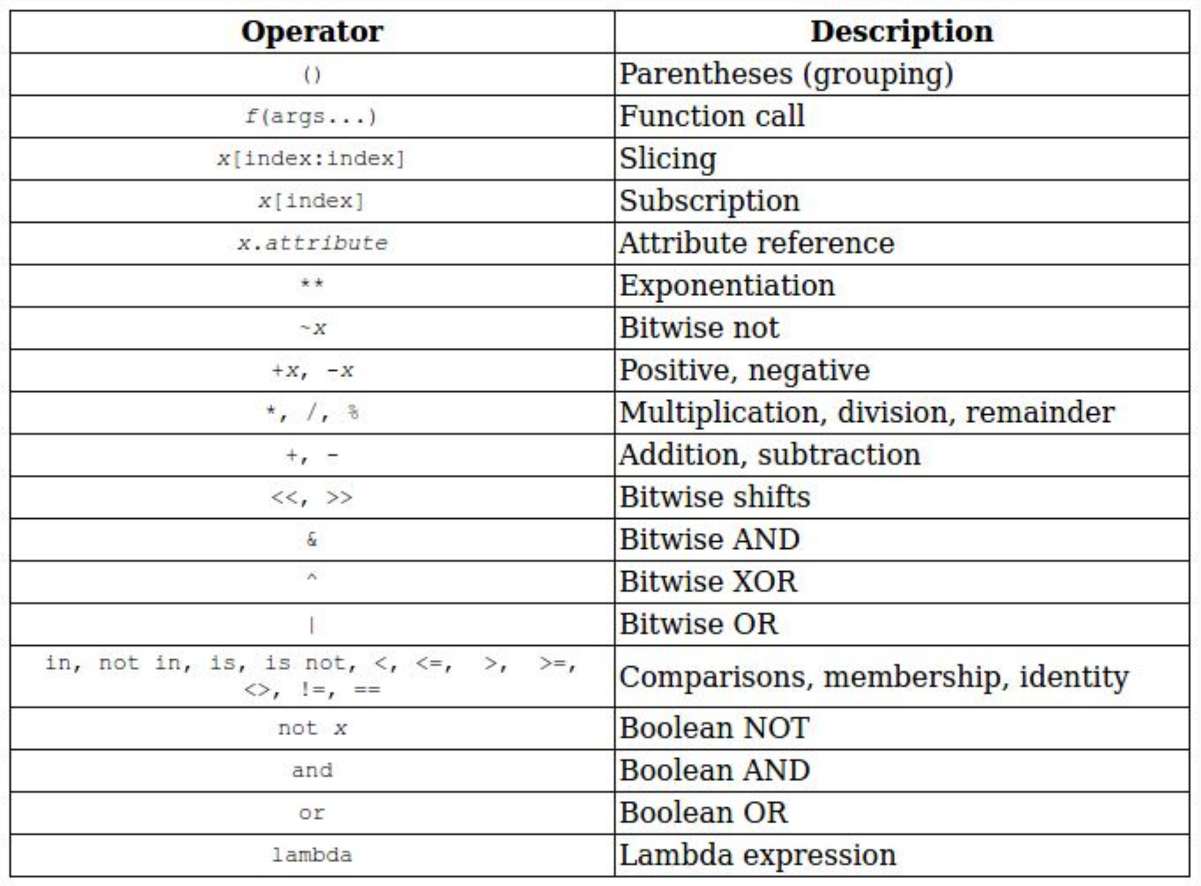
- Many thousands of predefined functions exist
- **Import** is used to access **functions** defined within **modules**
- It is also possible to define your own functions
- Functions specify **formal parameters** which can have defaults
- **Actual parameters**, commonly referred to as **arguments**, are passed when a function is called
- **Lambda** Expressions provide a mechanism of defining small anonymous style functions

List of built-in functions (no need to import)



		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Operator precedence in Python



Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression