

Week 9: Fundamentals of Python Programming II

POP77001 Computer Programming for Social Scientists

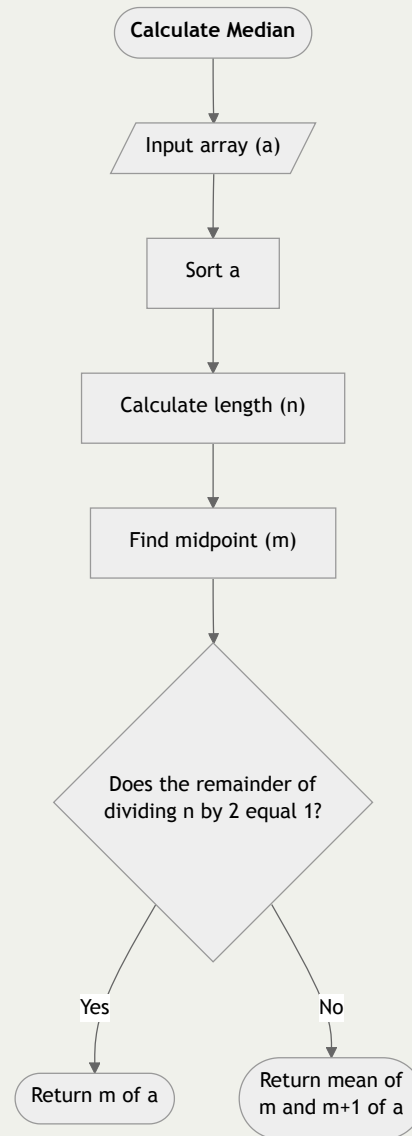
Tom Paskhalis

Overview

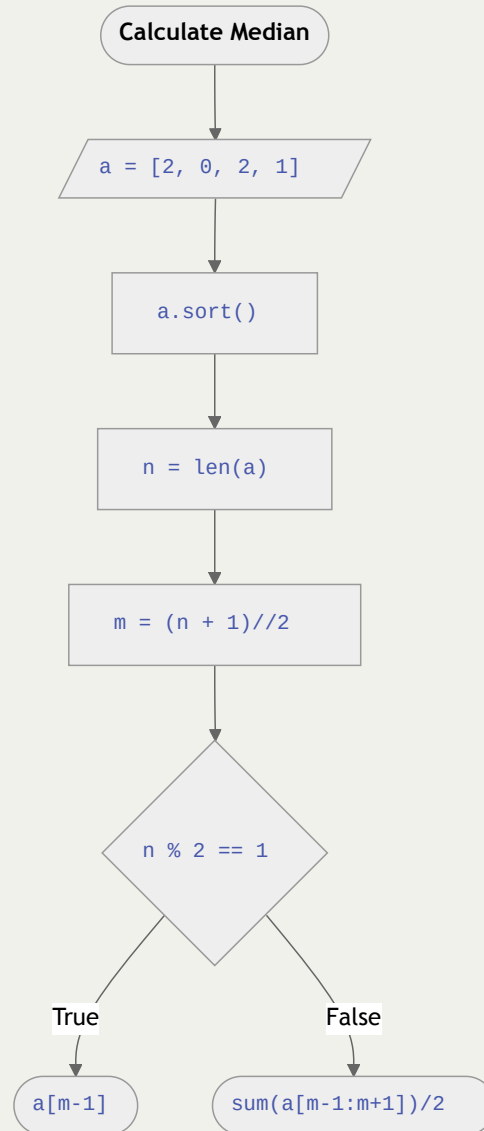
- Control flow
- Conditional statements
- Loops and iteration
- Iterables
- List comprehensions
- Functions

Control Flow

Algorithm Flowchart



Algorithm Flowchart (Python)



Calculate Median

```
1 a = [2, 0, 2, 1] # Input list
2 a.sort() # Sort list, note in-place modification
3 a
```

[0, 1, 2, 2]

```
1 n = len(a) # Calculate length of list 'a'
2 n
```

4

```
1 m = (n + 1)//2 # Calculate mid-point, // is operator for integer division
2 m
```

2

```
1 n % 2 == 1 # % (modulo) gives remainder of division
```

False

```
1 sum(a[m-1:m+1])/2 # Calculate median as the mean of the two numbers around
```

1.5

Control Flow in Python

- *Control flow* is the order in which statements are executed or evaluated
- Main ways of control flow in Python:
 - **Branching** (conditional) statements (e.g. `if`)
 - **Iteration** (*loops*) (e.g. `while`, `for`)
 - **Function calls** (e.g. `len()`)
 - **Exceptions** (e.g. `TypeError`)

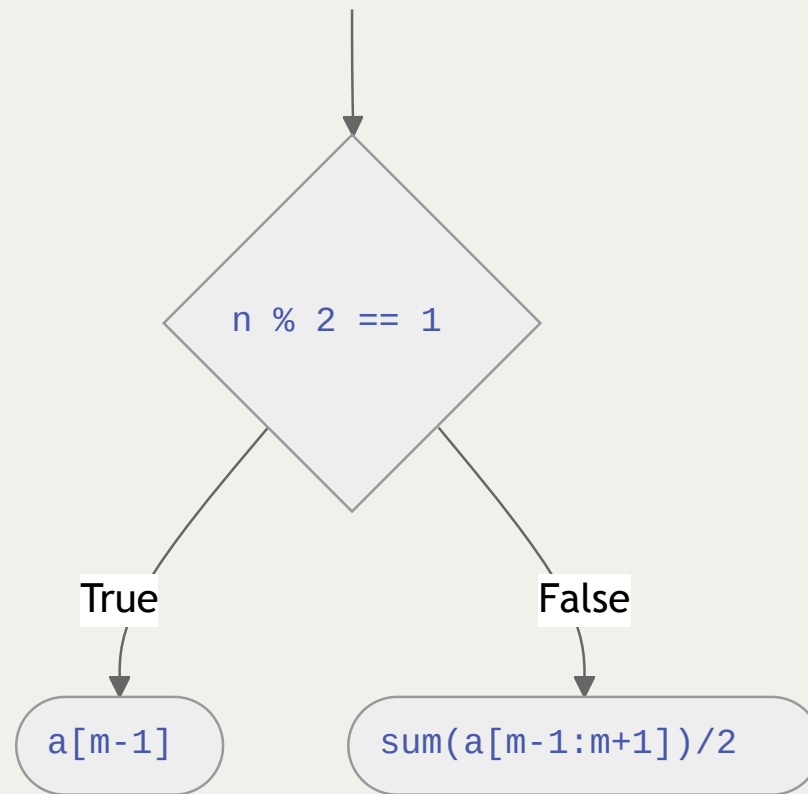


Extra

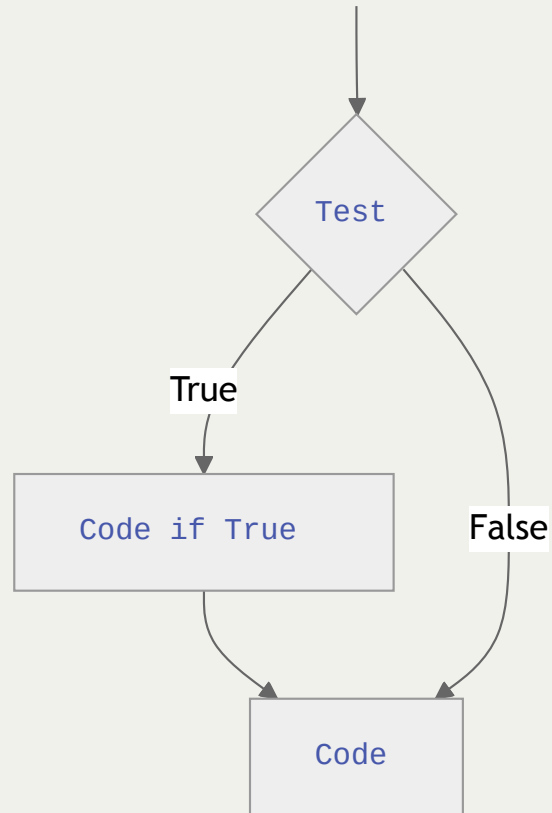
Python documentation on control flow

Conditional Statements

Branching Programs



Simple Conditional Statement



Basic Conditional Statement: **if**

- **if** - defines condition under which some code is executed

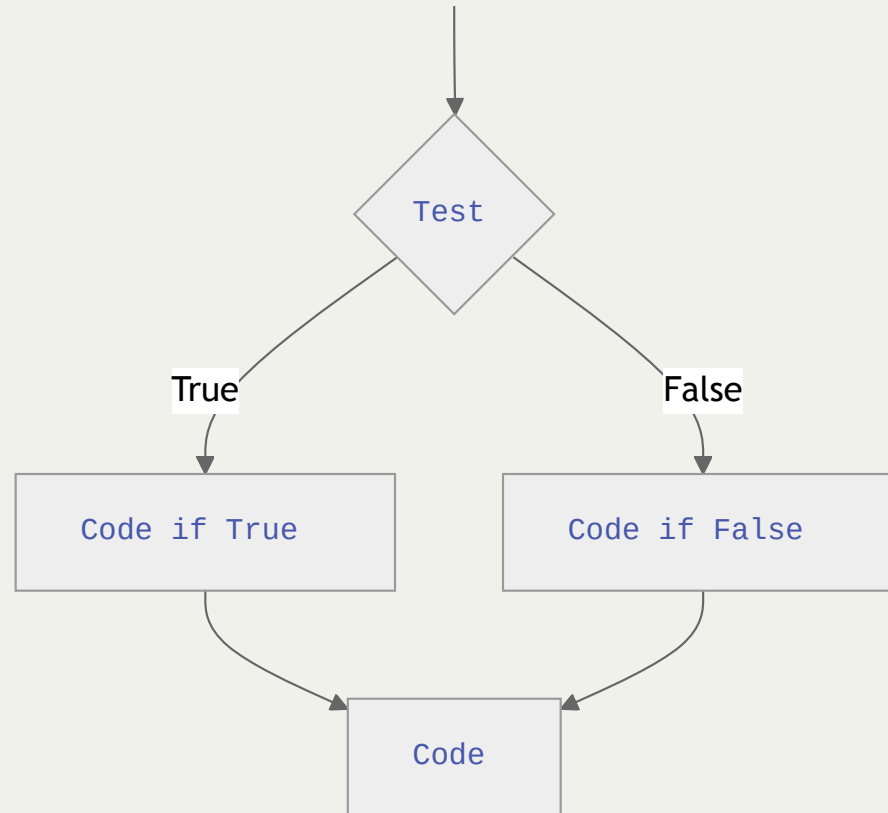
```
1 # Note that addition of a large value (100)
2 # has no effect on the median.
3 a = [2, 0, 2, 1, 100]
4 a.sort()
5 n = len(a)
6 m = (n + 1)//2
```

```
1 if n % 2 == 1:
2     a[m-1]
```

```
if <boolean_expression>:
    <some_code>
```

2

Complex Conditional Statements



if - else

- `if - else` - defines both condition under which some code is executed and alternative code to execute

```
1 a = [2, 0, 2, 1]
2 a.sort()
3 n = len(a)
4 m = (n + 1)//2
```

```
1 if n % 2 == 1:
2     a[m-1]
3 else:
4     sum(a[m-1:m+1])/2
```

```
if <boolean_expression>:
    <some_code>
else:
    <some_other_code>
```

1.5

if - elif - else

- `if - elif - ... - else` - defines both condition under which some code is executed and several alternatives

```
1 mark = 71
```

```
1 if mark >= 70:  
2     grade = "I"  
3 elif mark >= 60:  
4     grade = "II.1"  
5 elif mark >= 50:  
6     grade = "II.2"  
7 else:  
8     grade = "F"
```

```
if <boolean_expression>:  
    <some_code>  
elif <boolean_expression>:  
    <some_other_code>  
...  
...  
else:  
    <some_more_code>
```

```
1 grade
```

```
'I'
```

Indentation

- **Indentation** is semantically meaningful in Python.
- Visual structure of a program accurately represents its semantic structure.
- Tabs and spaces should not be mixed.
- E.g. Jupyter Notebook converts tabs to spaces by default.

Indentation in Python

```
1 x = 43
2 if x % 2 == 0:
3     'Even'
4     if x > 0:
5         'Positive'
6     else:
7         'Negative'
```

```
1 x = 43
2 if x % 2 == 0:
3     'Even'
4 if x > 0:
5     'Positive'
6 else:
7     'Negative'
```

'Positive'

Conditional Expressions

- Python supports **conditional expressions** as well as conditional statements

<expr1> if <test> else <expr2>

```
1 x = 42
2 y = 'even' if x % 2 == 0 else 'odd'
3 y
```

'even'

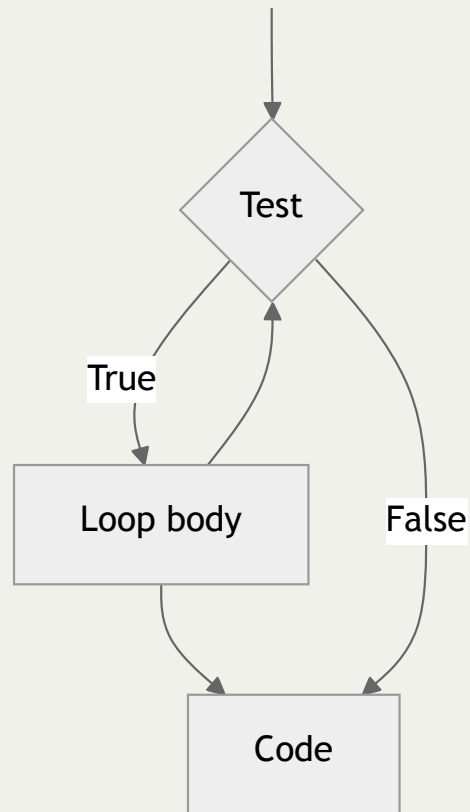
Which is analogous to:

```
1 x = 42
2 if x % 2 == 0:
3     y = 'even'
4 else:
5     y = 'odd'
6 y
```

'even'

Iteration

Loop



while

- **while** - defines a condition under which some code (loop body) is executed repeatedly

```
while <boolean_expression>:  
    <some_code>
```

```
1 # Calculate a factorial with decrementing function  
2 # E.g. 5! = 1 * 2 * 3 * 4 * 5 = 120  
3 x = 5  
4 factorial = 1  
5 while x > 0:  
6     factorial *= x # factorial = factorial * x  
7     x -= 1 # x = x - 1  
8 factorial
```

120

Iteration: **for**

- **for** - defines elements and sequence over which some code is executed iteratively

```
for <element> in <sequence>:  
    <some_code>
```

```
1 x = range(1, 6)  
2 factorial = 1  
3 for i in x:  
4     factorial *= i  
5 factorial
```

120

Iteration with Conditional Statements

```
1 # Find maximum value in a list with exhaustive enumeration
2 l = [3, 27, 9, 42, 10, 2, 5]
3 max_val = l[0]
4 for i in l[1:]:
5     if i > max_val:
6         max_val = i
7 max_val
```

42

range() Function

- `range()` function generates arithmetic progressions and is essential in `for` loops.
- In Python 3 `range()` is a **generator** function.
- It does not store all values at once (only start, stop and step).
- Rather it generates them on demand.

```
range(start, stop[, step])
```

```
1 r = range(3)
2 r
```

```
range(0, 3)
```

```
1 list(r)
```

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



[Python documentation for range\(\)](#)

[Python documentation for generator functions](#)

range() Function: Examples

```
1 l = [3, 27, 9, 42, 10, 2, 5]
2 for i in range(len(l)):
3     print(l[i], end = ' ')
```

3 27 9 42 10 2 5

```
1 l = [3, 27, 9, 42, 10, 2, 5]
2 s = []
3 for i in range(1, len(l), 2):
4     s.append(str(l[i]))
5 s
```

['27', '42', '2']

Iterables

- **Iterable** is an object that generates one element at a time within iteration.
- Formally, they are objects that have `__iter__` method, which return **iterator**.
- Some iterables are built-in (e.g. list, tuple, `range()`).
- But they can also be user-created.

Iteration over Multiple Iterables

- `zip()` function provides a convenient way of iterating over several sequences simultaneously.

```
1 l = [3, 27, 9, 42]
2 s = ['three', 'twenty seven', 'nine', 'forty-two']
3 for i, j in zip(l, s):
4     print(str(i) + ' - ' + j)
```

```
3 - three
27 - twenty seven
9 - nine
42 - forty-two
```



[Python documentation for zip\(\)](#)

Iteration over Dictionaries

- Iterating over a dictionary yields its keys.
- Alternatively, you can use one of the applicable methods to iterate over:
 - `keys()` - keys.
 - `values()` - values.
 - `items()` - key-value pairs.

```
1 d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

```
1 for i in d:  
2     i
```

```
'apple'  
'banana'  
'watermelon'
```

```
1 for k, v in d.items():  
2     print(k.upper(), int(v))
```

```
APPLE 150  
BANANA 120  
WATERMELON 3000
```

Iteration: **break** and **continue**

- **break** - terminates the loop in which it is contained
- **continue** - exits the iteration of a loop in which it is contained

```
1 for i in range(1,6):  
2     if i % 2 == 0:  
3         break  
4     print(i)
```

1

```
1 for i in range(1,6):  
2     if i % 2 == 0:  
3         continue  
4     print(i)
```

1

3

5

List Comprehensions

- **List comprehensions** provide a concise way to apply an operation to each element of a list.
- They offer a convenient and fast way of building list.
- Can have a nested structure (which affects legibility 📜).

```
[<expr> for <elem> in <iterable>]  
[<expr> for <elem> in <iterable> if <test>]  
[<expr> for <elem1> in <iterable1> for <elem2> in <iterable2>]
```

```
1 l = [0, 'one', 1, 2]
```

```
1 [x * 2 for x in l]
```

```
[0, 'oneone', 2, 4]
```

```
1 [x * 2 for x in l if type(x) == int]
```

```
[0, 2, 4]
```

```
1 [x.upper() for x in l if type(x) == str]
```

```
['ONE']
```



Extra

[Python documentation for list comprehensions](#)

Set and Dictionary Comprehensions

- Analogous to lists, sets and dictionaries have their own concise ways of iterating over them:

```
{<expr> for <elem> in <iterable> if <test>}  
{<key>: <value> for <elem1>, <elem2> in <iterable> if <test>}
```

```
1 o = {'apple', 'banana', 'watermelon'}  
2 {e[0].title() + ' - ' + e for e in o}
```

```
{'B - banana', 'A - apple', 'W - watermelon'}
```

```
1 d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}  
2 {k.upper(): int(v) for k, v in d.items()}
```

```
{'APPLE': 150, 'BANANA': 120, 'WATERMELON': 3000}
```

More on Iterations

- Always make sure that the terminating condition for a loop is properly specified.
- Nested loops can substantially slow down your program, try to avoid them.
- Use `break` and `continue` to shorten iterations.
- Consolidate several loops into one whenever possible.

Functions

Built-in & User-defined

- Python has many built-in functions: `len()`, `range()`, `zip()`.
- But its flexibility comes from functions defined by users.
- Many imported modules would contain their own functions.
- And many functions need to be implemented by the developer (i.e. you).

Function Definition

- Functions are defined using `def` statement.
- Variables are local to function definition in which they were assigned.
- Docstrings should be used to provide function overview (accessed with `help()`).

```
def <function_name>(arg_1, arg_2, ..., arg_n):  
    """<docstring>"""  
    <function_body>
```

```
1 def fun(arg):  
2     """This function does nothing"""  
3     pass # does nothing, but is required as 'def' statement cannot be empty
```



Python documentation on defining functions

Function Definition: Example

```
1 def calculate_median(lst):
2     """Calculates median
3
4     Takes list as input
5     Assumes all elements of list are numeric
6     """
7     lst.sort()
8     n = len(lst)
9     m = (n + 1)//2
10    if n % 2 == 1:
11        median = lst[m-1]
12    else:
13        median = sum(lst[m-1:m+1])/2
14    return median
```

Function Call

- Function is executed until:
 - Either `return` statement is encountered
 - There are no more expressions to evaluate
- Function call always returns a value:
 - Value of expression following `return`
 - `None` if no `return` statement

`<function_name>(arg_1, arg_2, ...)`

```
1 a = [2, 0, 2, 1]
2 calculate_median(a)
```

1.5

- Functions need to be defined before called

```
1 calculate_mean(a)
```

`NameError: name 'calculate_mean' is not defined`

Function Call: Example

```
1 def is_positive(num):  
2     if num > 0:  
3         return True  
4     elif num < 0:  
5         return False
```

```
1 res1 = is_positive(5)  
2 res2 = is_positive(-7)  
3 res3 = is_positive(0)
```

```
1 print(res1)
```

True

```
1 print(res2)
```

False

```
1 print(res3)
```

None

Function Arguments

- **Arguments** provide a way of giving input to a function.
- Arguments in function definition are sometimes called **parameters**.
- When a function is invoked (called) arguments are matched and bound to local variable names
- Python bounds function arguments in 2 ways:
 - by *position* (positional arguments)
 - by *keywords* (keyword arguments)
- A keyword argument cannot be followed by a non-keyword argument
- Keyword arguments are often used together with *default values*
- Supplying default values makes arguments *optional*

Function Arguments: Example

```
1 def format_date(day, month, year, reverse = True):  
2     if reverse:  
3         return str(year) + '-' + str(month) + '-' + str(day)  
4     else:  
5         return str(day) + '-' + str(month) + '-' + str(year)
```

```
1 format_date(10, 11, 2025)
```

'2025-11-10'

```
1 format_date(day = 10, month = 11, year = 2025)
```

'2025-11-10'

```
1 format_date(10, 11, 2025, False)
```

'10-11-2025'

```
1 format_date(day = 4, month = 11, year = 2024, False)
```

positional argument follows keyword argument (<string>, line 1)

Variable Number of Arguments

- `*` in function definition collects unmatched position arguments into a tuple.
- `**` collects keyword arguments into a dictionary.

```
1 def fun(*args, **kwargs):  
2     print(args)  
3     print(kwargs)
```

```
1 fun(  
2     1, 'x', [5, 6, 10],  
3     first = 1, second = 'x', third = [5, 6, 10]  
4 )
```

```
(1, 'x', [5, 6, 10])
```

```
{'first': 1, 'second': 'x', 'third': [5, 6, 10]}
```

Function Arguments: Hard Cases

- All types of arguments can be combined, although such cases are rare:

```
def <function_name>(
    arg_1, ..., arg_n, *args,
    kwarg_1, ..., kwarg_n, **kwargs
):
    <function_body>
```

```
1 def fun2(a, b, *args, c = False, **kwargs):
2     print(a, b, args, c, kwargs)
```

```
1 fun2(1, 'x', 20, 'cat', c = True, last = [10, 99])
```

```
1 x (20, 'cat') True {'last': [10, 99]}
```

Nested Functions

```
1 def which_integer(num):  
2     def even_or_odd(num):  
3         if num % 2 == 0:  
4             return 'even'  
5         else:  
6             return 'odd'  
7     if num > 0:  
8         eo = even_or_odd(num)  
9         return 'positive ' + eo  
10    elif num < 0:  
11        eo = even_or_odd(num)  
12        return 'negative ' + eo  
13    else:  
14        return 'zero'
```

```
1 which_integer(-43)
```

'negative odd'

```
1 even_or_odd(-43)
```

NameError: name 'even_or_odd' is not defined

Python Scope Basics

- Variables (aka names) exist in a **namespace**.
- This is where Python searches, when you refer to the object by its variable name.
- Location of first variable assignment determines its namespace (scope of visibility).

```
1 x = 5
```

```
1 def foo():  
2     x = 12  
3     return x
```

```
1 y = foo()  
2 print(y)
```

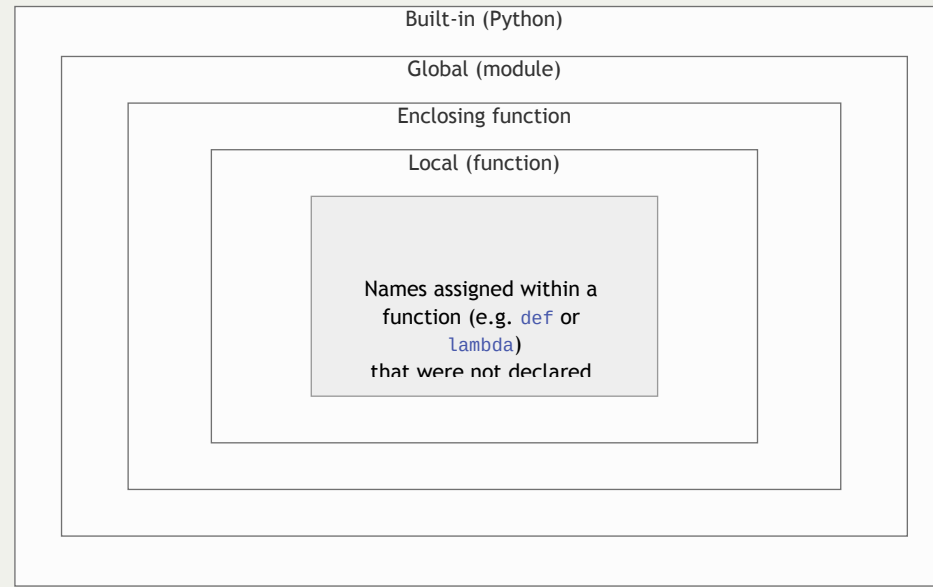
```
12
```

```
1 print(x)
```

```
5
```

Scoping Levels in Python

- Variables can be assigned in 3 different places, that correspond to 3 different scopes:
 - `local` to the function, if a variable is assigned inside `def`
 - `nonlocal` to nested function, if a variable is assigned in an enclosing `def`
 - `global` to the file (module), when a variable is assigned outside all `defs`



Lambda Functions

- Anonymous function objects can be created with `lambda` expression.
- It can appear in places, where defining function is not allowed by Python syntax.
- E.g. as arguments in higher-order functions, return values, etc.

```
lambda arg_1, arg_2, ... arg_n: <some_expression>
```

```
1 # function definition with `def` always binds function object to a name
2 def add_excl(s):
3     return s + '!'
4
5 add_excl('Function')
```

```
'Function!'
```

```
1 # typically, lambda function would not be assigned to a name
2 add_excl = lambda s: s + '!'
3
4 add_excl('Lambda')
```

```
'Lambda!'
```

Lambda Function: Example

```
1 import math
2
3 def make_scaler(scale = 'linear'):
4     if scale == 'linear':
5         return lambda x: x
6     elif scale == 'log':
7         return lambda x: math.log(x) if x > 0 else float('-inf')
8     else:
9         raise ValueError('Unknown scale')
```

```
1 # `log_scaler` is a function object that is yet to be invoked
2 log_scaler = make_scaler(scale = 'log')
```

```
1 log_scaler(10)
```

2.302585092994046

```
1 [log_scaler(x) for x in range(10)] # More Pythonic
```

[-inf, 0.0, 0.6931471805599453, 1.0986122886681098, 1.3862943611198906, 1.6094379124341003, 1.791759469228055, 1.9459101490553132, 2.0794415416798357, 2.1972245773362196]

```
1 # More functional in style, similar to R's:
2 # mapply(function(x) log(x), 0:9)
3 # unlist(Map(function(x) log(x), 0:9))
4 # but a lot more abstruse in Python
5 list(map(lambda x: math.log(x) if x > 0 else float('-inf'), range(10)))
```

[-inf, 0.0, 0.6931471805599453, 1.0986122886681098, 1.3862943611198906, 1.6094379124341003, 1.791759469228055, 1.9459101490553132, 2.0794415416798357, 2.1972245773362196]

Recursion



Reddit

Recursion in Programming

- Functions that call themselves are called **recursive** functions
- It consists of 2 parts that prevent it from being a circular solution:
 1. Base case, specifies the result of a special case
 2. General case, defines answer in terms of answer on some other input

Recursion: Example

- Factorial function:
 - Base case: $1! = 1$
 - General case: $n! = n * (n-1)!$

```
1 def factorial(x):  
2     """Calculates factorial of x!  
3  
4     Takes one integer as an input  
5     Returns the factorial of that integer  
6     """  
7     if x == 1:  
8         return x  
9     else:  
10        return x * factorial(x-1)
```

```
1 factorial(5)
```

120

Function Design Principles

- Function should have a single, cohesive purpose
 - Check if you could give it a short descriptive name
- Function should be relatively small
- Use arguments for input and return for output
 - Avoid writing to global variables
- Change mutable objects only if a caller expects it

Modules

- Module is `.py` file with Python definitions and statements.
- Program can access functionality of a module using `import` statement.
- Module is imported only once per interpreter session.
- Every module has its own namespace.

```
import <module_name>  
<module_name>.<object_name>
```

```
import <module_name> as <new_name>  
<new_name>.<object_name>
```

```
from <module_name> import <object_name>  
<object_name>
```

Module Import: Example

```
1 import statistics # Import all objects from module `statistics`  
2 from math import sqrt # Import only function `sqrt` from module `math`
```

```
1 fib = [0, 1, 1, 2, 3, 5]
```

```
1 statistics.mean(fib) # Mean
```

2

```
1 statistics.median(fib) # Median
```

1.5

```
1 sqrt(25) # Square root
```

5.0

Some Built-in Python Modules

Module	Description
<code>datetime</code>	Date and time types
<code>math</code>	Mathematical functions
<code>random</code>	Random numbers generation
<code>statistics</code>	Statistical functions
<code>os.path</code>	Pathname manipulations
<code>re</code>	Regular expressions
<code>pdb</code>	Python Debugger
<code>timeit</code>	Measure execution time of small code snippets
<code>csv</code>	CSV file reading and writing
<code>pickle</code>	Python object serialization (backup)



Extra

Python documentation for the Python Standard Library

Next

- Tutorial: Control flow and functions
- Assignment 3: Due at 12:00 on Monday, 17th November (submission on Blackboard)
- Next week: Data wrangling in Python