Week 9: Fundamentals of Python Programming II

POP77001 Computer Programming for Social Scientists

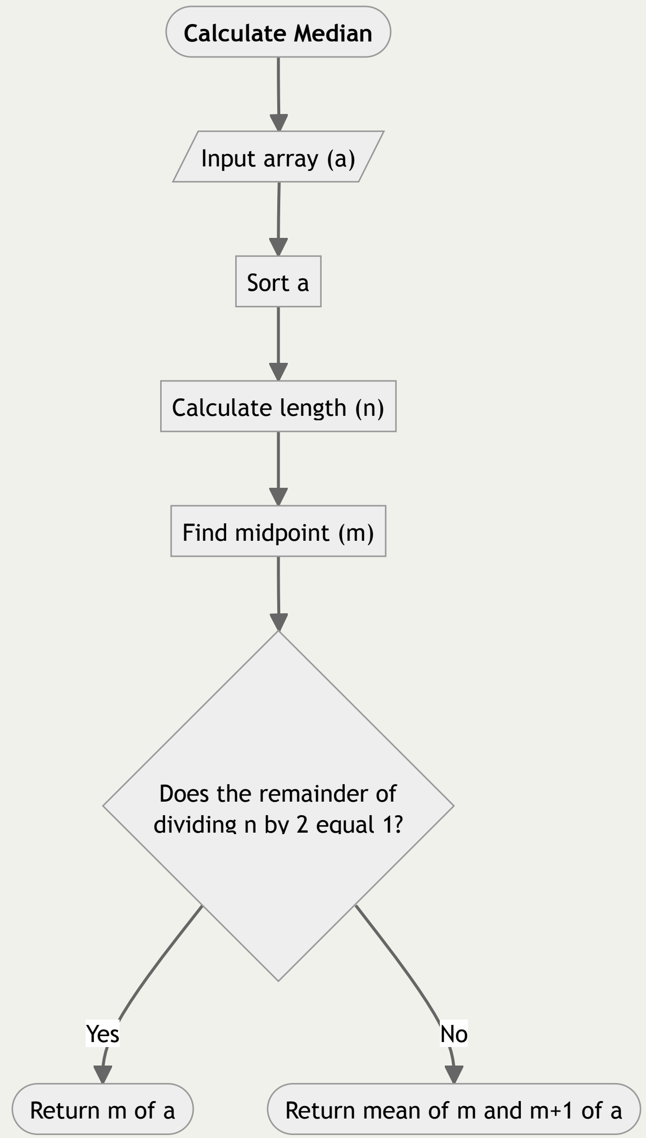
Tom Paskhalis

Overview

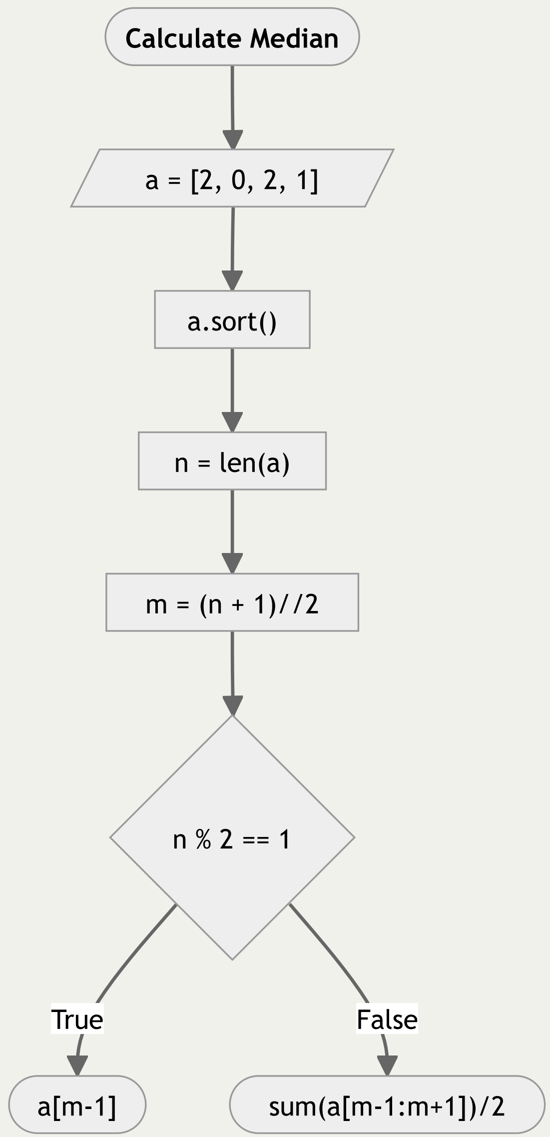
- Control flow
- Conditional statements
- Loops and iteration
- Iterables
- List comprehensions
- Functions

Control Flow

Algorithm Flowchart



Algorithm Flowchart (Python)



Calculate Median

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

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

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

```
4
```

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

```
2
```

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

```
False
```

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

```
1.5
```

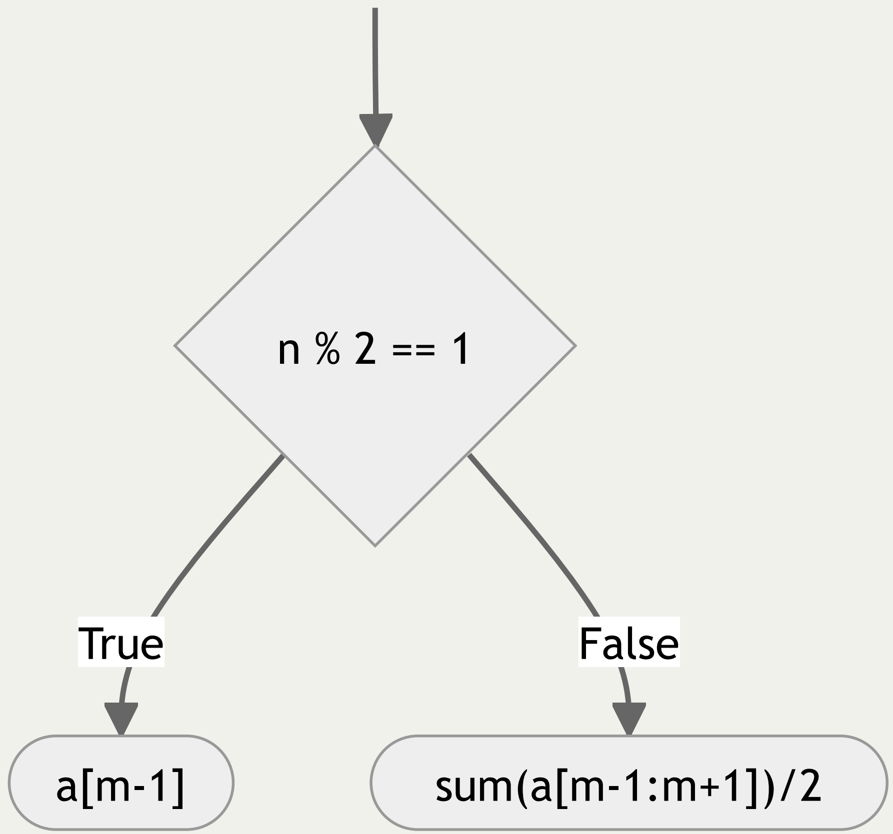
Control Flow in Python

- *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`)

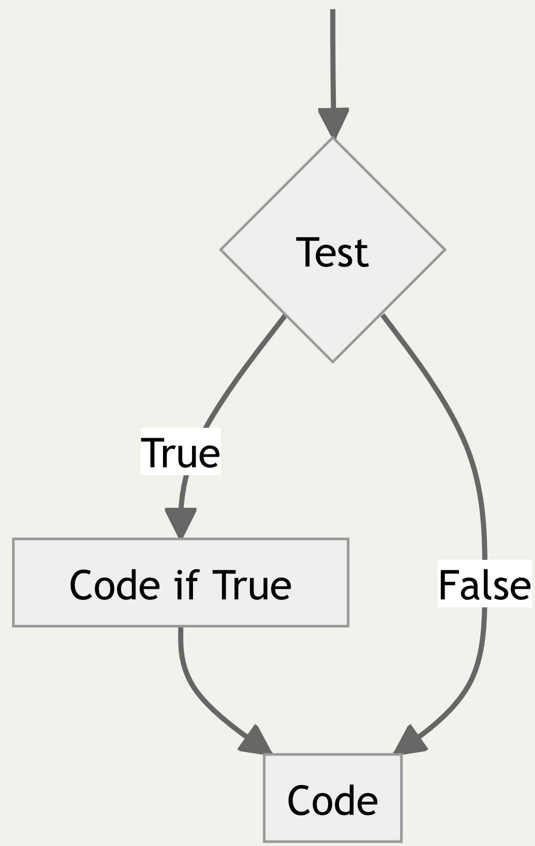
> **Extra**
>
> Python documentation on control flow

Conditional Statements

Branching Programs



Simple Conditional Statement



Basic Conditional Statement: `if`

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

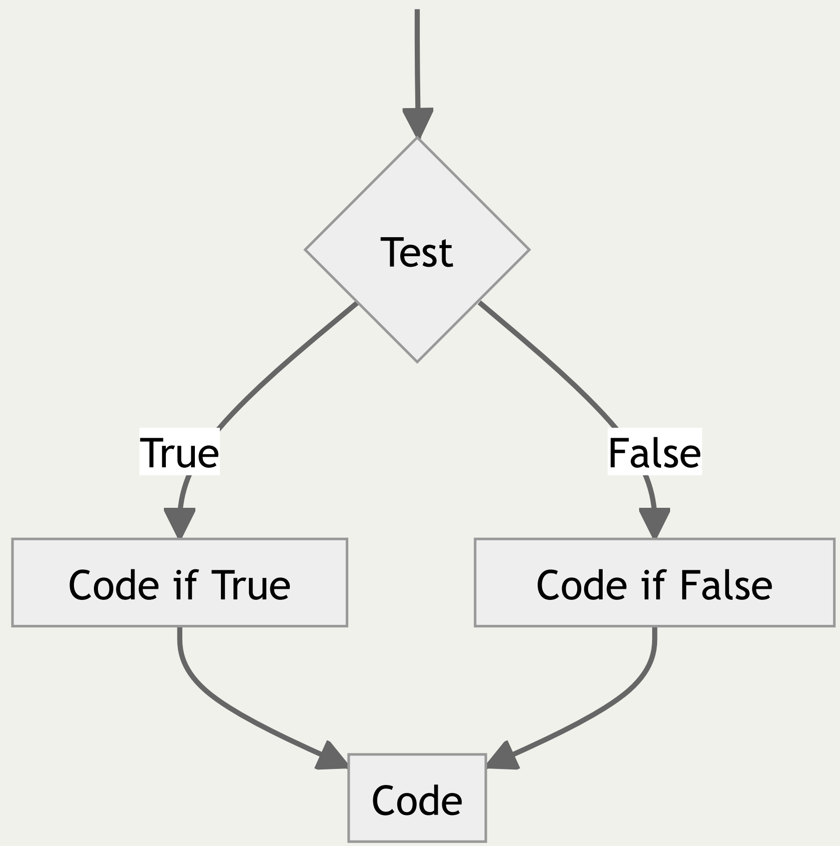
```python
# Note that addition of a large value (100)
# has no effect on the median.
a = [2, 0, 2, 1, 100]
a.sort()
n = len(a)
m = (n + 1)//2
```

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

```
2
```

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

Complex Conditional Statements



`if - else`

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

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

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

```
1.5
```

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

`if - elif - else`

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

```python
mark = 71
```

```python
if mark >= 70:
    grade = "I"
elif mark >= 60:
    grade = "II.1"
elif mark >= 50:
    grade = "II.2"
else:
    grade = "F"
```

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

```python
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

```python
x = 43
if x % 2 == 0:
    'Even'
    if x > 0:
        'Positive'
    else:
        'Negative'
```

```python
x = 43
if x % 2 == 0:
    'Even'
if x > 0:
    'Positive'
else:
    'Negative'
```

```
'Positive'
```

Conditional Expressions

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

```
<expr1> if <test> else <expr2>
```

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

```
'even'
```

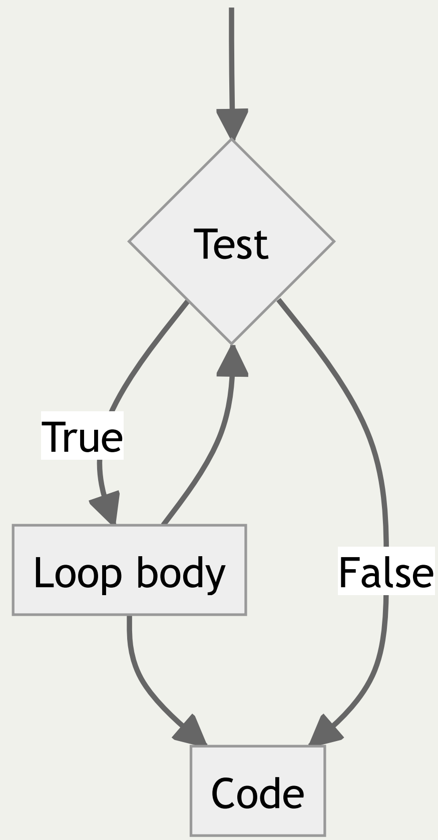
Which is analogous to:

```python
x = 42
if x % 2 == 0:
    y = 'even'
else:
    y = 'odd'
y
```

```
'even'
```

Iteration

Loop



`while`

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

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

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

```
120
```

Iteration: `for`

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

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

```python
x = range(1, 6)
factorial = 1
for i in x:
    factorial *= i
factorial
```

```
120
```

Iteration with Conditional Statements

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

```python
r = range(3)
r
```

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

```python
list(r)
```

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

Extra

Python documentation for range()

Python documentation for generator functions

`range()` Function: Examples

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

```
3 27 9 42 10 2 5
```

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

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

Iterables

- **Iterable** is an object that generates one element at a item 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.

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

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

Extra

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.

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

```
for i in d:
    i
```

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

```
for k, v in d.items():
    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

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

```
1
```

```python
for i in range(1,6):
    if i % 2 == 0:
        continue
    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>]
```

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

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

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

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

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

```python
[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>}
```

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

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

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

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

Extra

Python documentation on defining functions

Function Definition: Example

```
def calculate_median(lst):
    """Calculates median

    Takes list as input
    Assumes all elements of list are numeric
    """
    lst.sort()
    n = len(lst)
    m = (n + 1)//2
    if n % 2 == 1:
        median = lst[m-1]
    else:
        median = sum(lst[m-1:m+1])/2
    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, ...)
```

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

```
1.5
```

- Functions need to be defined before called

```
calculate_mean(a)
```

```
NameError: name 'calculate_mean' is not defined
```

Function Call: Example

```
def is_positive(num):
    if num > 0:
        return True
    elif num < 0:
        return False
```

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

```
print(res1)
```

```
True
```

```
print(res2)
```

```
False
```

```
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

```python
def format_date(day, month, year, reverse = True):
    if reverse:
        return str(year) + '-' + str(month) + '-' + str(day)
    else:
        return str(day) + '-' + str(month) + '-' + str(year)
```

```python
format_date(4, 11, 2024)
```

```
'2024-11-4'
```

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

```
'2024-11-4'
```

```python
format_date(4, 11, 2024, False)
```

```
'4-11-2024'
```

```python
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.

```python
def foo(*args):
    print(args)
```

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

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

```python
def foo(**kwargs):
    print(kwargs)
```

```python
foo(first = 1, second = 'x', third = [5,6,10])
```

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

Nested Functions

```python
def which_integer(num):
    def even_or_odd(num):
        if num % 2 == 0:
            return 'even'
        else:
            return 'odd'
    if num > 0:
        eo = even_or_odd(num)
        return 'positive ' + eo
    elif num < 0:
        eo = even_or_odd(num)
        return 'negative ' + eo
    else:
        return 'zero'
```

```python
which_integer(-43)
```

```
'negative odd'
```

```python
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).

```
x = 5
```

```
def foo():
    x = 12
    return x
```

```
y = foo()
print(y)
```

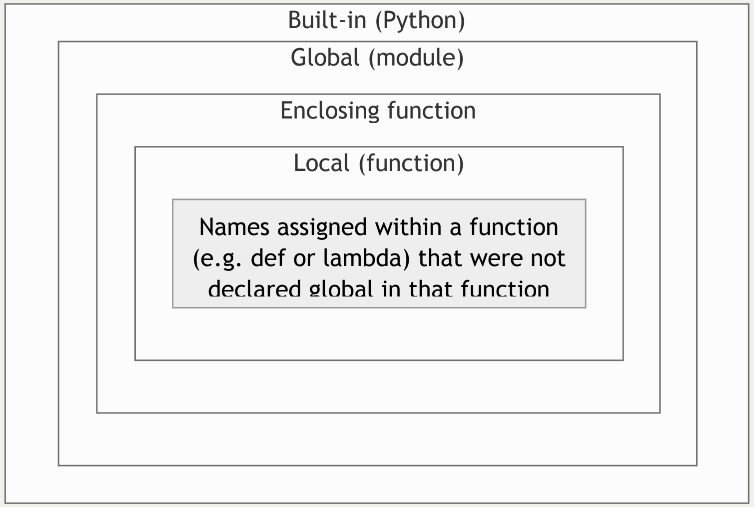
```
12
```

```
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 `def`s


Built-in (Python)
Global (module)
Enclosing function
Local (function)
Names assigned within a function (e.g. def or lambda) that were not declared global in that function

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>
```

```python
# function definition with `def` always binds function object to a name
def add_excl(s):
    return s + '!'

add_excl('Function')
```

```
'Function!'
```

```python
# typically, lambda function would not be assigned to a name
add_excl = lambda s: s + '!'

add_excl('Lambda')
```

```
'Lambda!'
```

Lambda Function: Example

```python
import math

def make_scaler(scale = 'linear'):
    if scale == 'linear':
        return lambda x: x
    elif scale == 'log':
        return lambda x: math.log(x) if x > 0 else float('-inf')
    else:
        raise ValueError('Unknown scale')
```

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

```python
log_scaler(10)
```

```
2.302585092994046
```

```python
[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]
```

```python
# More functional in style, similar to R's:
# mapply(function(x) log(x), 0:9)
# unlist(Map(function(x) log(x), 0:9))
# but a lot more abstruse in Python
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 if from being a circular solution:
  1. Base case, specifies the result of a special case
  2. General case, defines answer in terms of answer om some other input

Recursion: Example

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

```python
def factorial(x):
    """Calculates factorial of x!

    Takes one integer as an input
    Returns the factorial of that integer
    """
    if x == 1:
        return x
    else:
        return x * factorial(x-1)
```

```python
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

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

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

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

```
2
```

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

```
1.5
```

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

```
5.0
```

Some Built-in Python Modules

Module	Description
`datetime`	Date and time types
`math`	Mathematical functions
`random`	Random numbers generation
`statistics`	Statistical functions
`os.path`	Pathname manipulations
`re`	Regular expressions
`pdb`	Python Debugger
`timeit`	Measure execution time of small code snippets
`csv`	CSV file reading and writing
`pickle`	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, 11th November (submission on Blackboard)
- Next week: Data wrangling in Python