Week 9: Fundamentals of Python Programming II

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

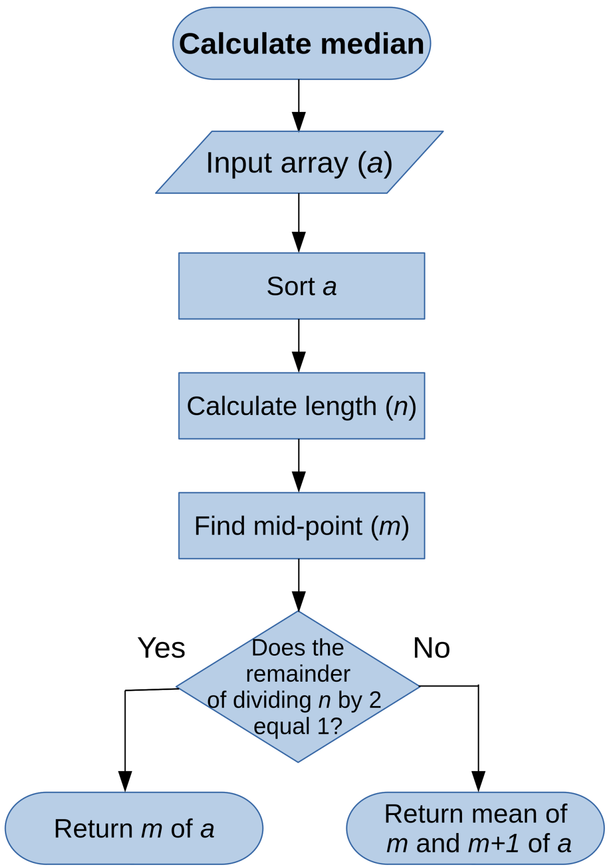
7 November 2022

Module website: tinyurl.com/POP77001

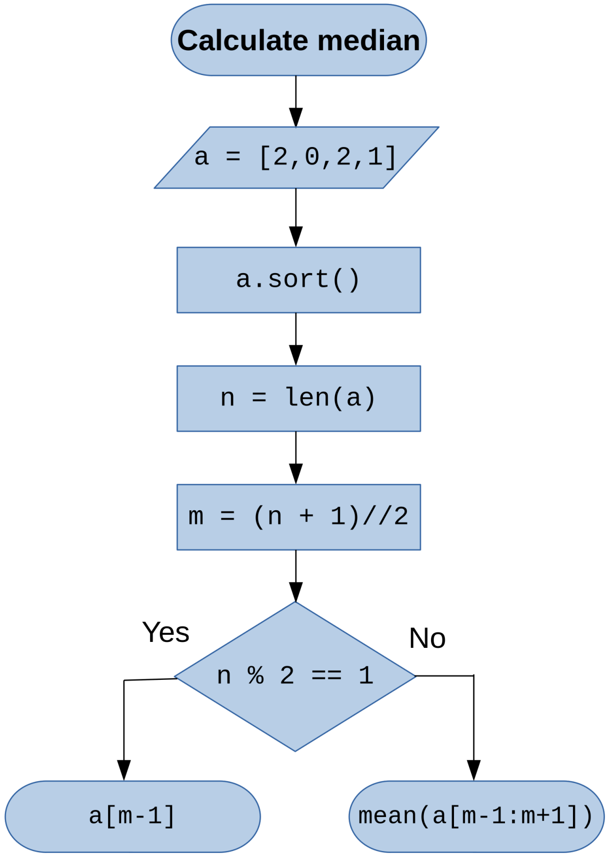
Overview

- Straight-line and branching programs
- Conditional statements
- Loops and iteration
- Iterables
- List comprehensions

Algorithm flowchart



Algorithm flowchart (Python)



Calculate median

Calculate median

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

Out[1]:
```
[0, 1, 2, 2]
```

Calculate median

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

Out[1]:
```
[0, 1, 2, 2]
```

In [2]:
```
n = len(a) # Calculate length of list 'a'
n
```

Out[2]:
```
4
```

Calculate median

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

Out[1]:
```
[0, 1, 2, 2]
```

In [2]:
```
n = len(a) # Calculate length of list 'a'
n
```

Out[2]:
```
4
```

In [3]:
```
m = (n + 1)//2 # Calculate mid-point, // is operator for integer divisi
m
```

Out[3]:
```
2
```

Calculate median

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

Out[1]:
```
[0, 1, 2, 2]
```

In [2]:
```
n = len(a) # Calculate length of list 'a'
n
```

Out[2]:
```
4
```

In [3]:
```
m = (n + 1)//2 # Calculate mid-point, // is operator for integer divisi
m
```

Out[3]:
```
2
```

In [4]:
```
n % 2 == 1 # Check whether the number of elements is odd, % (modulo) gi
```

Out[4]:
```
False
```

Calculate median

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

Out[1]:
```
[0, 1, 2, 2]
```

In [2]:
```
n = len(a) # Calculate length of list 'a'
n
```

Out[2]:
```
4
```

In [3]:
```
m = (n + 1)//2 # Calculate mid-point, // is operator for integer divisi
m
```

Out[3]:
```
2
```

In [4]:
```
n % 2 == 1 # Check whether the number of elements is odd, % (modulo) gi
```

Out[4]:
```
False
```

In [5]:
```
sum(a[m-1:m+1])/2 # Calculate median, as the mean of the two numbers ar
```

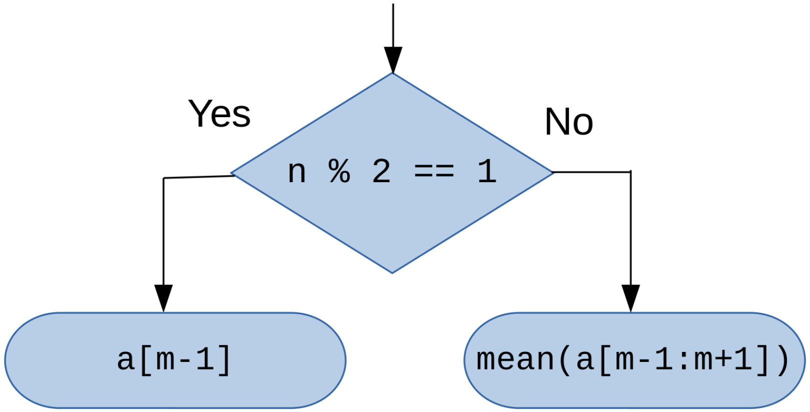
Out[5]:
```
1.5
```

Control flow in Python

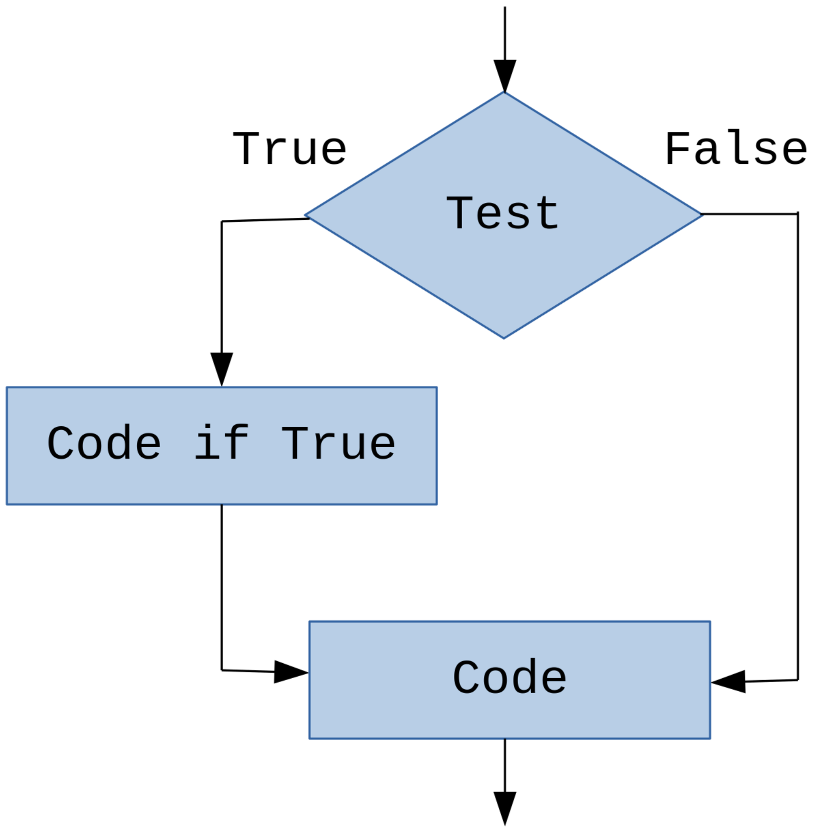
- *Control flow* is the order in which statements are executed or evaluated
- Main ways of control flow in Python:
    - *Branching* (conditional) statements (e.g. `if`)
    - *Iteration* (*loops*) (e.g. `while`)
    - *Function calls* (e.g. `len()`)
    - *Exceptions* (e.g. `TypeError`)

Extra: [Python documentation for control flow]

Branching programs



Conditional statements



Conditional statements: `if`

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

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

Conditional statements: `if`

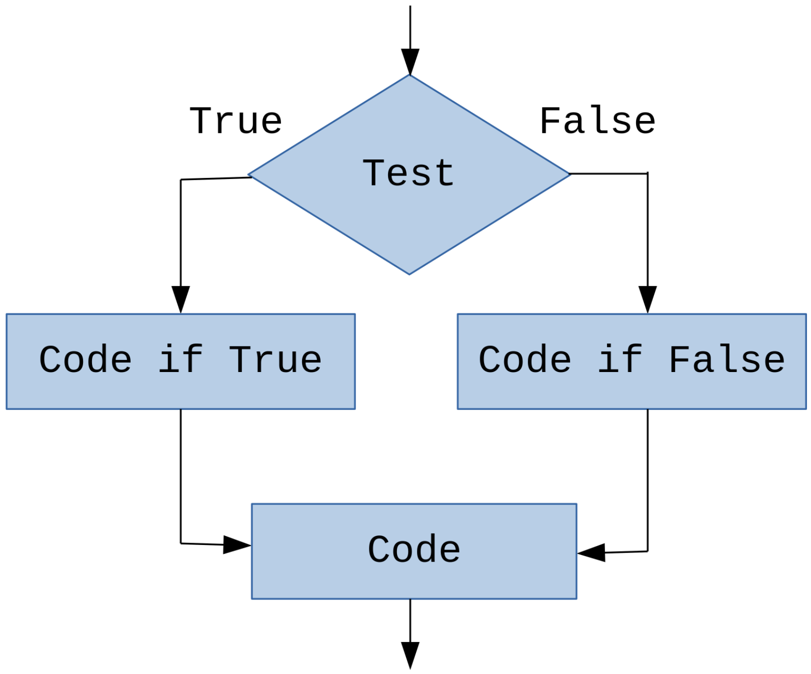
- `if` - defines condition under which some code is executed

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

In [6]:
```python
a = [2,0,2,1,100] # Note that addion of a large value (100) had no effe
a.sort()
n = len(a)
m = (n + 1)//2
if n % 2 == 1:
    print(a[m-1])
```

```
2
```

Conditional statements



Conditional statements: `if - else`

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

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

Conditional statements: `if - else`

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

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

In [7]:

```
a = [2,0,2,1]
a.sort()
n = len(a)
m = (n + 1)//2
if n % 2 == 1:
    print(a[m-1])
else:
    print(sum(a[m-1:m+1])/2)
```

```
1.5
```

Conditional statements: `if - elif - ... - else`

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

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

Example of longer conditional statement

Example of longer conditional statement

In [8]:

```
x = 42
if x > 0:
    print('Positive')
elif x < 0:
    print('Negative')
else:
    print('Zero')
```

```
Positive
```

Optimising conditional statements

- Parts of conditional statement are evaluated sequentially, so it makes sense to put the most likely condition as the first one

Optimising conditional statements

- Parts of conditional statement are evaluated sequentially, so it makes sense to put the most likely condition as the first one

In [9]:

```
num = float(input('Please, enter a number:')) # Ask for user input and
if num % 2 == 0:
    print('Even')
elif num % 2 == 1:
    print('Odd')
else:
    print('This is a real number')
```

```
Please, enter a number:43
Odd
```

Nesting conditional statements

- Conditional statements can be nested within each other
- But consider code legibility 📜, modularity ⚙️ and speed 🏎️

Nesting conditional statements

- Conditional statements can be nested within each other
- But consider code legibility 📜, modularity ⚙️ and speed 🏎️

In [10]:

```
num = int(input('Please, enter a number:')) # Ask for user input and ca
if num > 0:
    if num % 2 == 0:
        print('Positive even')
    else:
        print('Positive odd')
elif num < 0:
    if num % 2 == 0:
        print('Negative even') # Notice that odd/even checking appears
    else:
        print('Negative odd') # Consider abstracting this as a functior
else:
    print('Zero')
```

```
Please, enter a number:-43
Negative odd
```

Indentation

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

Indentation in Python code

Indentation in Python code

In [11]:

```
x = -43
if x % 2 == 0:
    print('Even')
    if x > 0:
        print('Positive')
    else:
        print('Negative')
```

Indentation in Python code

In [11]:
```
x = -43
if x % 2 == 0:
    print('Even')
    if x > 0:
        print('Positive')
    else:
        print('Negative')
```

In [12]:
```
x = -43
if x % 2 == 0:
    print('Even')
if x > 0:
    print('Positive')
else:
    print('Negative')
```

```
Negative
```

Conditional expressions

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

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

Conditional expressions

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

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

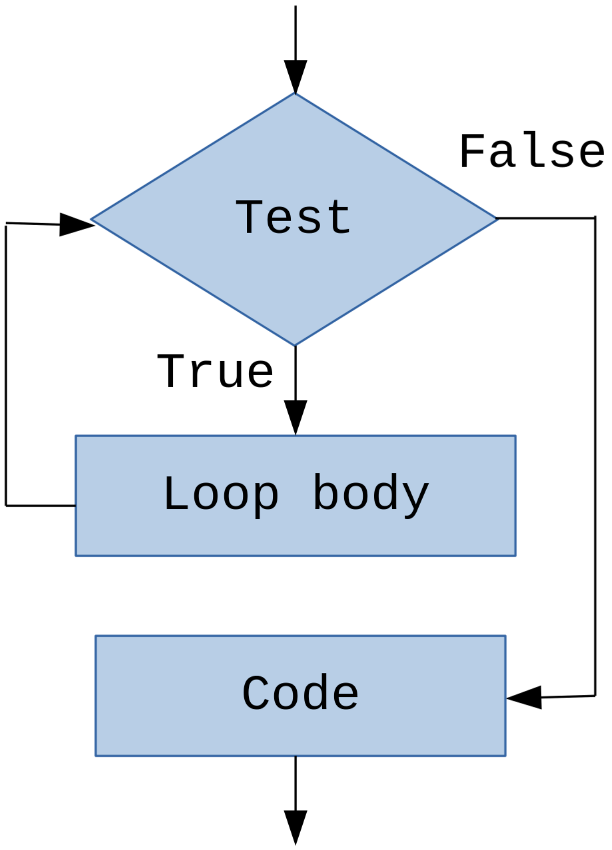
In [13]:

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

Out[13]:

```
'even'
```

Iteration (looping)



Iteration: `while`

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

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

Iteration: `while`

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

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

In [14]:

```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
```

Out[14]:

```
120
```

Iteration: `for`

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

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

Iteration: `for`

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

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

In [15]:

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

Out[15]:

```
120
```

Iteration with conditional statements

Iteration with conditional statements

In [16]:

```
# 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:
    if i > max_val:
        max_val = i
max_val
```

Out[16]:

```
42
```

`range()` function

- `range()` function generates arithmetic progressions and is essential in `for` loops

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

- The default values for `start` and `stop` are 0 and 1

Extra: Python documentation for range()

`range()` function

- `range()` function generates arithmetic progressions and is essential in `for` loops

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

- The default values for `start` and `stop` are 0 and 1

Extra: Python documentation for range()

In [17]:
```python
print(range(3))
print(list(range(3)))
```

```
range(0, 3)
[0, 1, 2]
```

`range()` function in `for` loops

`range()` function in `for` loops

In [18]:

```
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
```

`range()` function in `for` loops

In [18]:

```
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
```

In [19]:

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

Out[19]:

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

Iterables in Python

- *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

Extra: Python documentation for zip()

Iteration over multiple iterables

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

Extra: Python documentation for zip()

In [20]:

```
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
```

Iteration over dictionaries

- `items()` method allows to iterate over keys and values in a dictionary

Iteration over dictionaries

- `items()` method allows to iterate over keys and values in a dictionary

In [21]:

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

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

In [22]:
```
for i in range(1,6):
    if i % 2 == 0:
        break
    print(i)
```

```
1
```

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

In [22]:

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

```
1
```

In [23]:

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

```
1
3
5
```

Infinite loop

```
while True:
```

Source: Reddit

Infinite loops

- Loops that have no explicit limits for the number of iterations are called *infinite*
- They have to be terminated with a `break` statement (or Ctrl/Cmd-C in interactive session)
- Such loops can be unintentional (bug) or desired (e.g. waiting for user's input, some event)

Infinite loops

- Loops that have no explicit limits for the number of iterations are called *infinite*
- They have to be terminated with a `break` statement (or Ctrl/Cmd-C in interactive session)
- Such loops can be unintentional (bug) or desired (e.g. waiting for user's input, some event)

In [24]:

```
i = 1
while True:
    i += 1
    if i > 10:
        break
```

Infinite loops

- Loops that have no explicit limits for the number of iterations are called *infinite*
- They have to be terminated with a `break` statement (or Ctrl/Cmd-C in interactive session)
- Such loops can be unintentional (bug) or desired (e.g. waiting for user's input, some event)

In [24]:

```
i = 1
while True:
    i += 1
    if i > 10:
        break
```

In [25]:

```
i
```

Out[25]:

```
11
```

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 lists
- 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>]
```

Extra: Python documentation for list comprehensions

Examples of list comprehensions

Examples of list comprehensions

In [26]:

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

Examples of list comprehensions

In [26]:
```
l = [0, 'one', 1, 2]
```

In [27]:
```
[x * 2 for x in l]
```

Out[27]:
```
[0, 'oneone', 2, 4]
```

Examples of list comprehensions

```
In [26]: l = [0, 'one', 1, 2]

In [27]: [x * 2 for x in l]

Out[27]: [0, 'oneone', 2, 4]

In [28]: [x * 2 for x in l if type(x) == int]

Out[28]: [0, 2, 4]
```

Examples of list comprehensions

```
In [26]: l = [0, 'one', 1, 2]

In [27]: [x * 2 for x in l]

Out[27]: [0, 'oneone', 2, 4]

In [28]: [x * 2 for x in l if type(x) == int]

Out[28]: [0, 2, 4]

In [29]: [x.upper() for x in l if type(x) == str]

Out[29]: ['ONE']
```

Set and dictionary comprehensions

- Analogous to list, 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>}
```

Set and dictionary comprehensions

- Analogous to list, 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>}
```

In [30]:
```python
o = {'apple', 'banana', 'watermelon'}
{e[0].title() + ' - ' + e for e in o}
```

Out[30]:
```
{'A - apple', 'B - banana', 'W - watermelon'}
```

Set and dictionary comprehensions

- Analogous to list, 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>}
```

In [30]:
```python
o = {'apple', 'banana', 'watermelon'}
{e[0].title() + ' - ' + e for e in o}
```

Out[30]:
```
{'A - apple', 'B - banana', 'W - watermelon'}
```

In [31]:
```python
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
{k.upper(): int(v) for k, v in d.items()}
```

Out[31]:
```
{'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

Built-in and user-defined functions

- 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)

Defining functions

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

Defining functions

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

In [32]:

```python
def foo(arg):
    pass # does nothing, but is required as 'def' statement cannot be e
```

Defining functions

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

In [32]:

```python
def foo(arg):
    pass # does nothing, but is required as 'def' statement cannot be e
```

Extra: Python documentation on defining functions

Function definition

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

Function definition example

Function definition example

In [33]:

```
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
```

Calling functions

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

Calling functions

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

In [34]:
```
a = [2, 0, 2, 1]
calculate_median(a)
```

Out[34]:
```
1.5
```

Calling functions

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

In [34]:
```
a = [2, 0, 2, 1]
calculate_median(a)
```

Out[34]:
```
1.5
```

- Functions need to be defined before called

Calling functions

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

In [34]:
```
a = [2, 0, 2, 1]
calculate_median(a)
```

Out[34]:
```
1.5
```

- Functions need to be defined before called

In [35]:
```
calculate_mean(a)
```

```
----------------------------------------------------------------
-------------
NameError                                 Traceback (most recen
t call last)
Input In [35], in <cell line: 1>()
----> 1 calculate_mean(a)

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

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 call example

Function call example

In [36]:

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

Function call example

In [36]:

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

In [37]:

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

print(res1)
print(res2)
print(res3)
```

```
True
False
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

Function arguments example

In [38]:

```
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)
```

Function arguments example

In [38]:
```
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)
```

In [39]:
```
format_date(4, 10, 2021)
```

Out[39]:
```
'2021-10-4'
```

Function arguments example

In [38]:
```
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)
```

In [39]:
```
format_date(4, 10, 2021)
```

Out[39]:
```
'2021-10-4'
```

In [40]:
```
format_date(day = 4, month = 10, year = 2021)
```

Out[40]:
```
'2021-10-4'
```

Function arguments example

In [38]:
```
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)
```

In [39]:
```
format_date(4, 10, 2021)
```

Out[39]:
```
'2021-10-4'
```

In [40]:
```
format_date(day = 4, month = 10, year = 2021)
```

Out[40]:
```
'2021-10-4'
```

In [41]:
```
format_date(4, 10, 2021, False)
```

Out[41]:
```
'4-10-2021'
```

Function arguments example

In [38]:
```
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)
```

In [39]:
```
format_date(4, 10, 2021)
```

Out[39]:
```
'2021-10-4'
```

In [40]:
```
format_date(day = 4, month = 10, year = 2021)
```

Out[40]:
```
'2021-10-4'
```

In [41]:
```
format_date(4, 10, 2021, False)
```

Out[41]:
```
'4-10-2021'
```

In [42]:
```
format_date(day = 4, month = 10, year = 2021, False)
```

```
  Input In [42]
    format_date(day = 4, month = 10, year = 2021, False)
                                                       ^
SyntaxError: positional argument follows keyword argument
```

Functions with variable number of arguments

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

Functions with variable number of arguments

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

In [43]:
```
def foo(*args):
    print(args)
```

Functions with variable number of arguments

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

In [43]:
```
def foo(*args):
    print(args)
```

In [44]:
```
foo(1, 'x', [5,6,10])
```

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

Functions with variable number of arguments

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

In [43]:
```
def foo(*args):
    print(args)
```

In [44]:
```
foo(1, 'x', [5,6,10])
```

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

In [45]:
```
def foo(**kwargs):
    print(kwargs)
```

Functions with variable number of arguments

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

In [43]:
```
def foo(*args):
    print(args)
```

In [44]:
```
foo(1, 'x', [5,6,10])
```

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

In [45]:
```
def foo(**kwargs):
    print(kwargs)
```

In [46]:
```
foo(first = 1, second = 'x', third = [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>
```

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>
```

In [47]:

```python
def foo(a, b, *args, c = False, **kwargs):
    print(a, b, args, c, kwargs)
```

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>
```

In [47]:
```python
def foo(a, b, *args, c = False, **kwargs):
    print(a, b, args, c, kwargs)
```

In [48]:
```python
foo(1, 'x', 20, 'cat', c = True, last = [10, 99])
```

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

Nested functions

Nested functions

In [49]:

```
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'
```

Nested functions

In [49]:
```
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'
```

In [50]:
```
which_integer(-43)
```

Out[50]:
```
'negative odd'
```

Nested functions

In [49]:

```
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'
```

In [50]:

```
which_integer(-43)
```

Out[50]:

```
'negative odd'
```

In [51]:

```
even_or_odd(-43)
```

```
---------------------------------------------------------------
------------
NameError                                 Traceback (most recen
```

```
t call last)
Input In [51], in <cell line: 1>()
----> 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 looks it up, when you refer to an object by its variable name
- Location of first variable assignment determines its namespace (scope of visibility)

Python scope basics

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

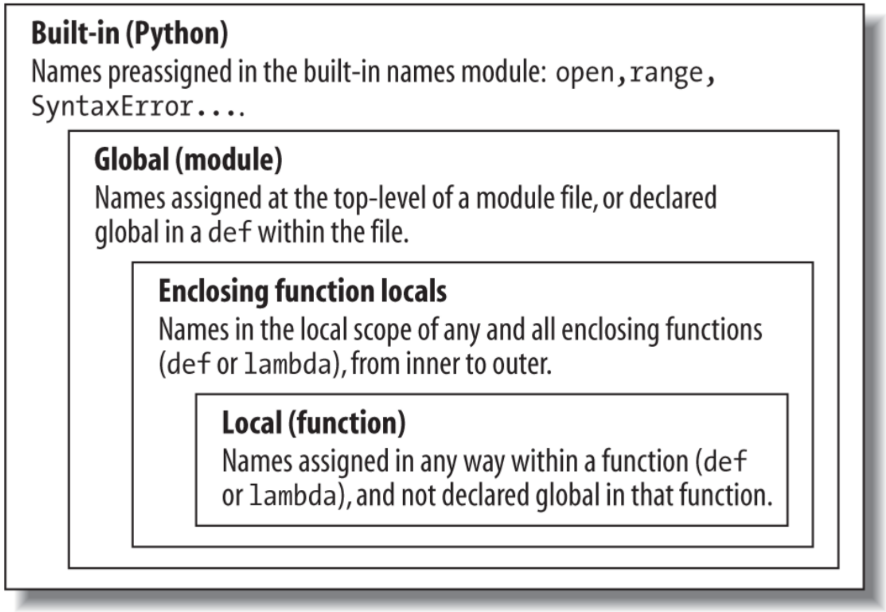
In [52]:

```python
x = 5
def foo():
    x = 12
    return x
y = foo()
print(y)
print(x)
```

```
12
5
```

Scoping levels

- 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



Lambda functions

- Apart from `def` statement, it is possible to generate function objects with `lambda` expression
- `lambda` allows creating anonymous function (returns function instead of assigning it to a name)
- Thus, it can appear in places, where defining function is not allowed by Python syntax
- E.g. as arguments in higher-order functions, return values

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

Lambda functions

- Apart from `def` statement, it is possible to generate function objects with `lambda` expression
- `lambda` allows creating anonymous function (returns function instead of assigning it to a name)
- Thus, it can appear in places, where defining function is not allowed by Python syntax
- E.g. as arguments in higher-order functions, return values

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

In [53]:
```python
def add_excl(s): # function definition always binds function object to
    return s + '!'

add_excl('Function')
```

Out[53]:
```
'Function!'
```

Lambda functions

- Apart from `def` statement, it is possible to generate function objects with `lambda` expression
- `lambda` allows creating anonymous function (returns function instead of assigning it to a name)
- Thus, it can appear in places, where defining function is not allowed by Python syntax
- E.g. as arguments in higher-order functions, return values

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

In [53]:
```python
def add_excl(s): # function definition always binds function object to
    return s + '!'

add_excl('Function')
```

Out[53]: 'Function!'

In [54]:
```python
add_excl = lambda s: s + '!' # typically, lambda function wouldn't be a
add_excl('Lambda')
```

Out[54]: 'Lambda!'

Lambda function examples

Lambda function examples

In [55]:

```
def add_five():
    return lambda x: x + 5

af = add_five()
```

Lambda function examples

In [55]:

```
def add_five():
    return lambda x: x + 5

af = add_five()
```

In [56]:

```
af # 'af' is just a function, which is yet to be invoked (called)
```

Out[56]:

```
<function __main__.add_five.<locals>.<lambda>(x)>
```

Lambda function examples

In [55]:
```
def add_five():
    return lambda x: x + 5

af = add_five()
```

In [56]:
```
af # 'af' is just a function, which is yet to be invoked (called)
```

Out[56]:
```
<function __main__.add_five.<locals>.<lambda>(x)>
```

In [57]:
```
af(10) # Here we call a function and supply 10 as an argument
```

Out[57]:
```
15
```

Lambda function examples

In [55]:
```
def add_five():
    return lambda x: x + 5

af = add_five()
```

In [56]:
```
af # 'af' is just a function, which is yet to be invoked (called)
```

Out[56]:
```
<function __main__.add_five.<locals>.<lambda>(x)>
```

In [57]:
```
af(10) # Here we call a function and supply 10 as an argument
```

Out[57]:
```
15
```

In [58]:
```
[x ** 2 for x in range(10)] # Could be faster, more 'pythonic'
```

Out[58]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Lambda function examples

In [55]:
```
def add_five():
    return lambda x: x + 5

af = add_five()
```

In [56]:
```
af # 'af' is just a function, which is yet to be invoked (called)
```

Out[56]:
```
<function __main__.add_five.<locals>.<lambda>(x)>
```

In [57]:
```
af(10) # Here we call a function and supply 10 as an argument
```

Out[57]:
```
15
```

In [58]:
```
[x ** 2 for x in range(10)] # Could be faster, more 'pythonic'
```

Out[58]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [59]:
```
list(map(lambda x: x**2, range(10))) # More functional in style, simila
```

Out[59]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Recursion

Source: 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:
    1. Base case: $1! = 1$
    2. General case: $n! = n * (n-1)!$

Recursion example

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

In [60]:
```
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)
```

Recursion example

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

In [60]:

```
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)
```

In [61]:

```
factorial(5)
```

Out[61]:

```
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 defitions 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

Module import example

In [62]:

```
import statistics # Import all objects (functions) from module 'statist
from math import sqrt # Import only function 'sqrt' from module 'math'

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

Module import example

In [62]:
```
import statistics # Import all objects (functions) from module 'statisti
from math import sqrt # Import only function 'sqrt' from module 'math'

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

In [63]:
```
statistics.mean(fib) # Mean
```

Out[63]:
```
2
```

Module import example

```
In [62]: import statistics # Import all objects (functions) from module 'statist
         from math import sqrt # Import only function 'sqrt' from module 'math'

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

In [63]: statistics.mean(fib) # Mean

Out[63]: 2

In [64]: statistics.median(fib) # Median

Out[64]: 1.5
```

Module import example

In [62]:
```
import statistics # Import all objects (functions) from module 'statist
from math import sqrt # Import only function 'sqrt' from module 'math'

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

In [63]:
```
statistics.mean(fib) # Mean
```

Out[63]:
```
2
```

In [64]:
```
statistics.median(fib) # Median
```

Out[64]:
```
1.5
```

In [65]:
```
sqrt(25) # Square root
```

Out[65]:
```
5.0
```

Some useful 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
- Next week: Data wrangling in Python