Functions and branching

Foundation of programming (CK0030)

Francesco Corona

FDP (cont)

Two fundamental and extremely useful programming concepts

- Functions, defined by the user
- Branching, of program flow
Functions

The term function has a wider meaning than a mathematical function.

Functions help avoid duplicating bits of code (puts all of them together)

- A strategy that saves typing and makes it easier to modify code

Functions are also used to split a long program into smaller pieces

Python has pre-defined functions (math.sqrt, range, len, math.exp, ...)

~ We discuss how to define own functions
Math functions as Python functions

We construct a Python function that evaluates a mathematical function.

\[ F(C) = \frac{9}{5}C + 32 \]

The function \( F \) takes \( C \) (as its input argument)

```python
def F(C):
    return (9.0/5)*C + 32
```

It returns value \((9.0/5)*C + 32\) (\(F(C)\)) as output.

Math functions as Python functions (cont.)

All Python functions begin with `def`, this is followed by the function name

\( \sim \) Inside parentheses, a comma-separated list of function arguments

\( \sim \) The argument acts as a standard variable inside the function.

The statements to be performed inside the function must be indented.

After the function, it is common (not necessary) to `return` a value

\( \sim \) The function output value is sent out of the function.

Math functions as Python functions (cont.)

Example

The function name is \( F \)

\[ F(C) = \frac{9}{5}C + 32 \]

There is only one `input` argument \( C \)

```python
def F(C):
    return (9.0/5)*C + 32
```

The return value is computed as \((9.0/5)*C + 32\) (it has no name)

- It is the evaluation of \( F(C) \) (implicitly \( F(C) \))
Math functions as Python functions (cont.)

The def line (function name and arguments) is the function header
The indented statements are the function body

1. def F(C):
2. return (9.0/5)*C + 32 # Function header
3. # Function (mini) block

The return often (not necessarily) associates with the function name

1. def F(C):
2. return (9.0/5)*C + 32 # F (C) #
3. # Function (mini) block

The value returned from F(C) is an object
~ Specifically, it is a float object

The call F(C) can be placed anywhere in a code
• A float must be valid

Example

1. ########################################################################
2. def F(C):
3. return (9.0/5)*C + 32 # T conversion function #
4. ########################################################################

To use a function, we must call or invoke it with input arguments
~ The function will process the input arguments
~ As a result, it will return an output value

We (may need to) store the result value in a variable

1. temp1 = F(15.5) # Store return value as variable (temp1)
2. a = 10 # Given input argument 'a' (value 10)
3. temp2 = F(a) # Store return value as variable (temp2)
4. print F(a+1) # Given input argument 'a+1' (value 10+1)
5. # Print return value to screen (no storing)
6. sum_temp = F(10) + F(20) # Two calls to get two output values #
7. # Combine output values and store
Math functions as Python functions (cont.)

Example

Consider the usual list `degrees` of temperatures in degrees Celsius.

We are interested in computing a list of corresponding Fahrenheits.

We want to use function `F`, in a list comprehension.

```python
degrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
Fdegrees = [F(C) for C in degrees]
```

How to use this new function?

```python
def F2(C):
    F_value = (9.0/5)*C + 32
    return '%.1f degrees Celsius correspond to %.1f degrees Fahrenheit' % (C, F_value)
```

The code contains a function `F(C)` and a `while` loop.

```python
while C <= 50:
    print '%.1f %.1f' % (C, F(C))
    C = C + 4
```

The code contains a function `F(C)` and a `while` loop.

```python
while C <= 50:
    print '%.1f %.1f' % (C, F(C))
    C = C + 4
```
Math functions as Python functions (cont.)

Programmers must understand the sequence of statements in a program
- There are excellent tools that help build such understanding
- A debugger and/or the Online Python Tutor

A debugger should be used for all sorts of programs, large and small
- Online Python Tutor is an educational tool (small programs)

Go to Online Python Tutor (link/click me), copy and paste your code
Use the ‘forward’ button to advance, one statement at a time
- Observe the sequence of operations
- Observe the evolution of variables
- Observe, observe, observe, ...

Local and global variables

Local variables are variables that are defined within a function
Local variables are invisible outside functions

Definition:

Local variables are variables that are defined within a function
Local variables are invisible outside functions

Example:

Consider the following function

```python
def F2(C):
    F_value = (9.0/5)*C + 32
    return '%.1f degrees Celsius correspond to
        %.1f degrees Fahrenheit' % (C, F_value)
```

Consider a simple function call

```python
>>> s1 = F2(21)
>>> s1
'21.0 degrees Celsius correspond to 69.8 Fahrenheit'
```

In function F2(C), variable F_value is a local variable
- It is inside a function

A local variable does not ‘exist’ outside the function
- (It cannot be accessed and used for computations)
Local and global variables (cont.)

```python
def F2(C):
    # F_value = (9.0/5)*C + 32
    # \( \frac{9}{5} \) degrees Celsius correspond to \
    # \( \frac{5}{9} \) degrees Fahrenheit \( C, F_{value} \)
    F_value = (9.0/5)*C + 32
    return '%.1f degrees Celsius correspond to '
    \( \frac{5}{9} \) degrees Fahrenheit \( C_{value} \) (%.1f degrees Fahrenheit)'
```

The (main) program around function F2(C) is not aware of variable F_value

```python
>>> c1 = 37.5
>>> s2 = F2(c1)
>>> F_value
NameError: name 'F_value' is not defined
```

Remark

Local variables are created inside a function

~ They are destroyed when leaving the function

Also input arguments are local variables

~ They cannot be accessed outside the function

Local and global variables (cont.)

Definition

Variables defined outside the function are global variables

Global variables are accessible everywhere in a program

~ Also from inside a function

Example

Consider the input argument to function F2, variable C

~ Variable C is a local variable

```python
def F2(C):
    # F_value = (9.0/5)*C + 32
    # \( \frac{9}{5} \) degrees Celsius correspond to \
    # \( \frac{5}{9} \) degrees Fahrenheit \( C, F_{value} \)
    F_value = (9.0/5)*C + 32
    return '%.1f degrees Celsius correspond to '
    \( \frac{5}{9} \) degrees Fahrenheit \( C_{value} \) (%.1f degrees Fahrenheit)
```

We cannot access variable C outside the function

```python
>>> c1 = 37.5
>>> s2 = F2(c1)
>>> F_value
NameError: name 'F_value' is not defined
>>> C
NameError: name 'C' is not defined
```
We ask the function to write out its variables

- Two local variables F_value, C
- A global variable r

The example illustrates also that there are two different variables C
Local and global variables (cont.)

The two variables C
  • C local variable exists only when the program flow is inside F3
  • C global variable is defined outside in the main (an int object)

Local variables hide/shade global variables
~ This is important

Local and global variables (cont.)

Consider the single-line piece of code

```
print sum
```

There are no local variables in the first line of code

Python then searches for a global variable, sum
~ It cannot find any

Python then checks among all built-in functions
~ It finds a built-in function with name sum
~ print sum returns <built-in function sum>

Local and global variables (cont.)

Consider now this three-line piece of code

```
print sum
sum = 500
print sum
```

The second line binds global name sum to an int object
At accessing sum in print statement, Python searches global variables
• Still no local variables are present
• It finds the one just defined

The printout becomes 500

Local and global variables (cont.)

Remark

Technically, global variable C can (still) be accessed as globals()['C']
  • This practice is deprecated

Avoid local and global variables with the same name at the same time!

The general rule, when there are variables with the same name
  • Python first looks up the name among local variables
  • Then, it searches among global variables
  • And, then among built-in functions
Local and global variables (cont.)

Consider the following piece of code

```python
print sum  # sum is a built-in Python function
sum = 500  # rebind name sum to an int object
print sum  # sum is a global variable

#############################################################

def myfunc():
    # sum is a local variable
    print sum

#############################################################

def f1(x):
    # a is a global variable
    a = 21
    return a*x + b

# show 20
print a

#############################################################

def f2(x):
    # a is declared global
    global a
    a = 21  # the global a is changed
    return a*x + b

f1(3); print a  # 20 is printed
f2(3); print a  # 21 is printed
```

Note that within function f1, \(a = 21\) creates a local variable \(a\)

- This does not change the global variable \(a\)

Call `myfunc()` invokes a function where `sum` is a local variable

`print sum` makes Python first search among local variables

- `sum` is found there, the printout is 3
- `(The printout is not 500, the value of global variable `sum`)` Value of local variable `sum` is returned, added to 1, and `print` function

- The `int` object is then bound to global variable `sum` (value 4)

Final `print sum` searches `global variables`, it finds one (value 4)

Remark

The values of global variables can be accessed inside functions

- Though their values cannot be changed
- Unless the variable is declared as global

### Multiple arguments

**Functions**

- Mathematical functions as Python functions
- Local and global variables
- Multiple arguments
- Function argument is a global variable
- Beyond math functions
- Multiple return values
- Global variables
- Keyword arguments
- Doc strings
- Functions as arguments to functions
- The main program
- Lambda functions
- Branching
- IF-ELSE blocks
- Inline IF-tests
Multiple arguments

Functions $F(C)$ and $F_2(C)$ are functions of one single variable $C$

- Both functions take one input argument ($C$)

Yet, functions can have as many input arguments as needed

- Need to separate the input arguments by commas (,)

Example

Consider the mathematical function

\[
y(t) = v_0 t - \frac{1}{2} g t^2
\]

$g$ is a fixed constant and $v_0$ is a physical parameter that can vary

Mathematically, function $y$ is a function of one variable, $t$

- The function values also depend on the value $v_0$
- To evaluate $y$, we need values for both $t$ and $v_0$

\[
y(t) = v_0 t - \frac{1}{2} g t^2 \quad | \quad v_0 = 5 \text{ [ms$^{-1}$]}, \quad g = 9.81 \text{ [ms$^{-2}$]}
\]
Multiple arguments (cont.)

\[ y(t) = v_0 t - \frac{1}{2} g t^2 \]

A natural implementation would be a function with two arguments

```python
# yfunc has two arguments: t and v0
def yfunc(t, v0):
    # g = 9.81
    return v0*t - 0.5*g*t**2
```

Within the function `yfunc`, arguments `t` and `v0` are local variables
- `g` is also a local variable

Suppose that we omit the `argument=value` syntax for all arguments
- The sequence of the arguments is no longer important
- `(We can place v0 before t)`

Suppose that we omit the `argument` part
- Then, it is important to remember that the sequence of arguments in the call must match (exactly) the sequence of arguments in the header

Advantages deriving from writing `argument=value` in the call
- Reading and understanding the statement is easier

Multiple arguments (cont.)

Suppose that we are interested in the function \( y(t) = v_0 t - \frac{1}{2} gt^2 \)

- \( v_0 = 6 \text{ [ms}^{-1}\text{]} \)
- \( t = 0.1 \text{ [s]} \)

```python
# Suppose that we are interested in the function y(t) = v0*t - 0.5*g*t**2
# v0 = 6 [ms^{-1}]
# t = 0.1 [s]

def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

# Arguments are given in the call
y1 = yfunc(0.1, 6)
y2 = yfunc(t=0.1, v0=6)
y3 = yfunc(t=0.1, v0=6)
y4 = yfunc(v0=6, t=0.1)
```

Remark

Consider `argument=value` arguments

They must appear AFTER all the arguments where only value is provided

```python
## yfunc(0.1, v0=6) is correct
~ yfunc(t=0.1, 6) is illegal
```
Multiple arguments (cont.)

Consider the case in which $yfunc(0.1, 6)$ or $yfunc(v0=6, t=0.1)$ is used. The arguments are automatically initialised as local variables.

The `exist` within the function

Initialisation is the same as assigning values to variables

```python
1 t = 0.1
2 v0 = 6.
3
4 def yfunc(t, v0):
5     #
6     g = 9.81
7     return v0*t - 0.5*g*t**2
8     #
9
10 g=9.81
```

Such statements are not visible in the code.
### Function argument v global variable

#### Example

Consider the following construction

```python
# Global variable v0
>>> v0 = 5.0
>>> vfunc(0.6)
1.2342
```

Variable `v0` is interpreted as a **global variable**

- It needs to be initialised outside function `yfunc`
- Before we attempt to call `yfunc`

#### Beyond math functions

- Any set of statements to be repeatedly executed under slightly different circumstances is a candidate for a Python function
Beyond math functions

Example

We want to make a list of numbers

Starting from some value (start) and stop at some other value (stop)

- We have given increments (inc)

Consider using variables start=2, stop=8, and inc=2

This would produce numbers 2, 4, 6, and 8

Beyond math functions (cont.)

```python
# Beyond math functions (cont.)
range(start, stop, inc) does not make the makealst function redundant

# makelst can generate real numbers
# range can only generate integers
```

Multiple returns

Functions

- Function `makelst` has three arguments: `start`, `stop`, and `inc`
- Inside the function, the arguments become local variables
- Also `value` and `result` are local variables

In the surrounding program (main), we define one variable, `mylist`

- Variable `mylist` is a global variable

Multiple returns

```python
>>> mylist = makelst(0, 100, 0.2)
>>> print mylist
# It will print the sequence 0.0, 0.2, 0.4, 0.6... 98.8, 100
```

Multiple returns

```python
def makelst(start, stop, inc):
    value = start
    result = []
    while value <= stop:
        result.append(value)
        value = value + inc
    return result

>>> mylist = makelst(0, 100, 0.2)
>>> print mylist
# It will print the sequence 0.0, 0.2, 0.4, 0.6... 98.8, 100
```
**Multiple returns**

**Example**

Suppose that we are interested in a function \( y(t) \) and its derivative \( y'(t) \)

\[
y(t) = v_0 t - \frac{1}{2}gt^2 \\
y'(t) = v_0 - gt
\]

Suppose that we want to get both \( y(t) \) and \( y'(t) \) from function \( \text{yfunc} \)

```python
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt
```

We included both calculations, then we separated variables in the `return` statement.

**Multiple returns (cont.)**

We can use the function `yfunc` in the production of a formatted table

- Values of \( t \), \( y(t) \) and \( y'(t) \)

```python
>>> position, velocity = yfunc(0.6, 3)
```

Format `%.10g` prints a real number as compactly as possible
- Whether in decimal or scientific notation
- Within a field of width 10 characters

The minus sign (−) after the percentage sign (%)
- Prints a number that is left-adjusted
  (Important for creating nice-looking columns)
Multiple returns (cont.)

Consider the following function

```python
def f(x):
    return x, x**2, x**4
```

Three objects are returned as output arguments

```bash
>>> a = f(2)
>>> a
(2, 4, 16)  # Stored as a tuple
```

Remark

Functions returning multiple (comma-separated) values returns a tuple
Summation

Suppose we are interested in creating a function to calculate the sum

\[
L(x;n) = \sum_{i=1}^{n} \frac{1}{i} \left( \frac{x}{1 + x} \right)^i
\]

Example

To compute the sum, a loop and add terms to an accumulation variable

- We performed a similar task with a while loop

Summations with integer counters (like \(i\)) are normally (often) implemented by a for-loop over the \(i\) counter (we performed also this task)

1. \(x = 0\)
2. for \(i \in \text{range}(1, n+1)\):
3. \(z += i^2\)
Summation (cont.)

\[ L(x; n) = \sum_{i=1}^{n} \frac{1}{x + i} \]

We want to embed the computation of the sum in a Python function

\[ \sim x \text{ and } n \text{ are the input arguments} \]
\[ \sim \text{ The sum } s \text{ is the return output} \]

```python
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += 1.0/i * (x/(1.0+x))**i
    return s
```

The size of the terms decreases with \( n \)

\[ \sim \text{ The first neglected term is bigger than all remaining terms} \]
\[ \sim \text{ (those calculated for } n+2, n+3, \ldots) \]

Yet, it is not necessarily bigger than their sum

\[ \sim \text{ We may use this term as a crude estimate of the error} \]

Summation (cont.)

It can be shown that \( L(x; n) \) is an approximation to \( \ln(1 + x) \)

\[ \sim \lim_{n \to \infty} L(x; n) = \ln(1 + x) \]

Instead of having \( L \), return only the value of the sum \( s \), it would be also interesting to return additional information on the approximation error

```python
for i in range(1, n+1):
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
value_of_sum = s
exact_error = log(1+x) - value_of_sum
```

We return the exact error (we calculate the `log` function by `math.log`)

```python
value, approximate_error, exact_error = L2(x, 100)
```
Consider the construction of a table of the accuracy of function \( L(x; n) \) ~ It is an approximation to \( \ln(1 + z) \)

```python
# Example

# Define L2(x, n):
def L2(x, n):
    z = 0
    for i in range(1, n+1):
        z += (1.0/(i)) * (x/(1.0+x))**(i)
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
    value_of_sum = z
    from math import log
    exact_error = log(1+z) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# Define table(x):
print '{\nx=%g, ln(1+x)=%g\n\x'} % (x, log(1+x))

for x in [1, 2, 10, 100, 500]:
    value, next, error = L2(x, n)
    print '{\nx=%g, ln(1+x)=%g\n\x'} % (x, log(1+x))
    print ' \n\x'} % (x, value, next, error)

########################################################################
>>> table(10)
1: x=10, ln(1+x)=2.39789
2: x=20, ln(1+x)=2.39789
3: x=100, ln(1+x)=2.39789
4: x=1000, ln(1+x)=2.39789

Value of sum, first neglected term, actual error
```

Sometimes a function can be defined to perform a set of statements

- Without necessarily computing objects returned to calling code
- The function without return values

In such situations, the return statement is not needed
Functions and branching

No returns (cont.)

```python
>>> table(100)
```

UFC/DC FdP - 2017.1

Functions

Mathematical functions as Python functions
Local and global variables
Multiple arguments
Function arguments as global variables
Beyond math
Functions
Multiple returns
Summation
No return
Keyword arguments
Doc strings
Functions as arguments to functions,
The main program
Lambda functions
Branching
IF-ELSE blocks
Inline IF-tests

No returns (cont.)

```python
def table(x):
    n=500 6.34928
    n=100 5.08989
    n=10 2.919
    n=2 1.498
    n=1 0.999001
    s=0
    for i in range(1, n+1):
        x = (1.0/i)*x/(1.0+x)**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error
```
### Functions and branching

**UFC/DC FdP - 2017.1**

#### Functions
- Mathematical functions as Python functions
- Local and global variables
- Multiple arguments
- Function arguments v global variable
- Beyond math functions
- Multiple returns
- Summation

#### Branching
- IF-ELSE blocks
- Lambda functions

---

#### Keyword arguments

The input arguments of a function can be assigned a default value

~ These arguments can be left out in the call

This is how a such a function may be defined

```python
# Keyword arguments
Keyword arguments

# The standard way to test if an object obj is set to None or not reads
if obj is None:
    ...
if obj is not None:
    ...

~ The is operator tests if two names refer to the same object
~ The == tests checks if the contents of two objects are the same
```

1. `a = 1`  
2. `b = a`  
3. `a is b`  
4. `True`  
5. `c = 1.0`  
6. `a is c`  
7. `False`  
8. `a == c`  
9. `True`

---

#### Functions as arguments to functions

The main program

Lambda functions

---

#### Branching

IF-ELSE blocks

---

#### No returns (cont.)

The standard way to test if an object `obj` is set to `None` or not reads

```python
if obj is None:
    ...
if obj is not None:
    ...
```

~ The `is` operator tests if two names refer to the same object

~ The `==` tests checks if the contents of two objects are the same

```python
>>> a = 1
>>> b = a
>>> a is b
# and b refer to the same object
True
>>> c = 1.0
>>> a is c
# a and c do not refer to the same object
False
>>> a == c
# a and c are mathematically equal
True
```

---

#### Keyword arguments (cont.)

First args (here, `arg1` and `arg2`) are *ordinary/positional arguments*

Last two args (`kwarg1` and `kwarg2`) are *keyword/named arguments*

Each *keyword argument* has a name and an associated a default value
Keyword arguments (cont.)

**Example**

```python
>>> somefunc('Hello', [1, 2])
Hello [1, 2] True 0
```

```python
>>> somefunc('Hello', [1, 2], kwarg1='Hi')
Hello [1, 2] Hi 0
```

```python
>>> somefunc('Hello', [1, 2], kwarg2='Hi')
Hello [1, 2] True Hi
```

```python
>>> somefunc('Hello', [1, 2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi
```

•**Keyword arguments (cont.)**

**Remark**

**Keyword arguments** must be listed AFTER positional arguments.

Suppose that **ALL input arguments** are explicitly referred to **(name=value)**.

The sequence is not relevant, **positional** and **keyword** can be mixed up.

```python
>>> somefunc(kwarg2='Hello', arg1='Hi', arg2=1, kwarg1=6)
```

```python
Hi [1, 2] 6 Hello
```

•**Keyword arguments (cont.)**

**Example**

Consider some function of \( t \) also containing some parameters \( A \), \( a \) and \( \omega \)

\[
f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)
\]

We have,

![Graph of f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)](image)

We implement \( f \) as function of independent variable \( t \), ordinary argument.

We set parameters \( A \), \( a \), and \( \omega \) as **keyword arguments** with default values.

```python
from math import pi, exp, sin
```
Keyword arguments (cont.)

We can call function \( f \) with only argument \( t \) specified

```python
>>> v1 = f(0.2)
```

Some of the other possible function calls

- \( f(x, A=1, a=1, \omega=2\pi) \)
- \( f(0.2, \omega=1) \)
- \( f(1, A=5, \omega=\pi, a=\pi^2) \)
- \( f(0.2, 0.5, 1, 1) \)

Example

Consider \( L(x; n) \) and functional implementations \( L_1(x, n) \) and \( L_2(x, n) \)

\[
L(x; n) = \sum_{i=1}^{n} \frac{1}{i} \left( \frac{x}{1+x} \right)^i
\]

We can now specify a minimum tolerance value \( \varepsilon \) for the accuracy

\[
\sim \frac{1}{n}
\]

We can use the first neglected term as an estimate of the accuracy
- Add terms as long as the absolute value of next term is greater than \( \varepsilon \)
Keyword arguments (cont.)

There is a convention to augment functions with some documentation

- The documentation string, known as a doc string
- A short description of the purpose of the function
- It explains what arguments and return values are
- Placed after the def funcname: line of definition

Doc strings are usually enclosed in triple double quotes ****

There is a convention to augment functions with some documentation

- The documentation string, known as a doc string
- A short description of the purpose of the function
- It explains what arguments and return values are
- Placed after the def funcname: line of definition

Doc strings are usually enclosed in triple double quotes ****

This allows the string to span several lines

There is a convention to augment functions with some documentation

- The documentation string, known as a doc string
- A short description of the purpose of the function
- It explains what arguments and return values are
- Placed after the def funcname: line of definition

Doc strings are usually enclosed in triple double quotes ****

This allows the string to span several lines

The output from calling table2(10)

```python
>>> table2(10)
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 9.70e-08, n=142
epsilon: 1e-10, exact error: 9.30e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

The epsilon estimate is about ten times smaller than the exact error

- regardless of the size of epsilon

epsilon follows the exact error over many orders of magnitude

We may view epsilon as a valid indication of error size

There is a convention to augment functions with some documentation

- The documentation string, known as a doc string
- A short description of the purpose of the function
- It explains what arguments and return values are
- Placed after the def funcname: line of definition

Doc strings are usually enclosed in triple double quotes ****

This allows the string to span several lines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2F(C)</td>
<td>Convert Celsius degrees (C) to Fahrenheit.</td>
</tr>
<tr>
<td>C: Input argument, temperature in Celsius return: Temperature in Fahrenheit</td>
<td></td>
</tr>
<tr>
<td>9.0/5*C + 32</td>
<td></td>
</tr>
</tbody>
</table>

Example

```python
def C2F(C):
    # Convert Celsius degrees (C) to Fahrenheit.
    # C: Input argument, temperature in Celsius
    return (9.0/5)*C + 32
```
Doc strings (cont.)

Example

```python
def line(x0, y0, x1, y1):
    ""
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = ax + b that goes
    through two points (x0, y0) and (x1, y1).
    ""
    x0, y0: a point on the line (floats)
    x1, y1: another point on the line (floats)
    return: coefficients a, b (floats) for the line (y=ax+b).
```

```
  a = (y1-y0)/float(x1-x0)
  b = y0-a*x0
  return a, b
```

To extract doc strings from source code use `funcname.__doc__`

```python
print line.__doc__
```

```
Compute the coefficients a and b in the mathematical
expression for a straight line y = ax + b that goes
through two points (x0, y0) and (x1, y1).

x0, y0: a point on the line (float objects).
x1, y1: another point on the line (float objects).
return: coefficients a, b (float objects) for the line (y=ax+b).
```

Doc strings often contain interactive sessions, from the Python shell

- They are used to illustrate how the function can be used

```python
>>> a, b = line(1, -1, 4, 3)
>>> a
1.3333333333333333
>>> b
-2.3333333333333333
```
The usual convention in Python

- **Function arguments** represent input data to the function
- **Returned objects** represent output data from function

A general Python function

```python
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io6; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7
```

- `i1`, `i2`, `i3` are positional arguments, input data
- `io4` and `io5` are positional arguments, input and output data
- `i6` and `io7` are keyword arguments, input and input/output data
- `o1`, `o2`, and `o3` are computed in the function, output data

Functions as arguments to functions

We can have functions to be used as arguments to other functions

A math function $f(x)$ may be needed for specific Python functions

- **Numerical root finding**
  - Solve $f(x) = 0$, approximately
- **Numerical differentiation**
  - Compute $f'(x)$, approximately
- **Numerical integration**
  - Compute $\int_a^b f(x)dx$, approximately
- **Numerical solution of differential equations**
  - Compute $x(t)$ from $\frac{dx}{dt} = f(x)$, approximately

In such functions, function $f(x)$ can be used as input argument ($f$)
This is straightforward in Python and hardly needs any explanation:

- In most other languages, special constructions must be used.
- Transfer a function to another function as argument.

### Example

**Compute the 2nd-order derivative of some function \( f(x) \), numerically**

\[
f''(x) \approx \frac{f(x - h) - 2f(x) + f(x + h)}{h^2}
\]

where \( h \) is a small number.

A Python function for the task:

```python
def diff2nd(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/h**2
    return r
```

This function is, like other input arguments, a name for a function object.

### Asymptotically, the approximation of the derivative get more accurate

- as \( h \to 0 \)

We show this property by making a table of the second-order derivatives:

- \( g(t) = t^{-6} \) at \( t = 1 \) as \( h \to 0 \)

```python
for k in range(1,15):
    h = 10**(-k)
    diff2nd(g, 1, h)
    print '%.4f' % (h, d2g)
```
Functions as arguments to functions (cont.)

The exact answer is \( g''(t = 1) = 42 \)

- For small \( h \) rounding errors blow up and destroy accuracy
- Switching from standard floating-point numbers (\texttt{float}) to numbers with arbitrary high precision (\texttt{module decimal}) solves the problem

Computations start returning very inaccurate results for \( h < 10^{-8} \)

The main program

In programs with \texttt{functions}, a part of the program is called \texttt{main}

- It is the collection of all statements outside the \texttt{functions}
- It is the collection of all statements outside the \texttt{functions}

Example

```python
from math import *
# In main

def f(x):
    # A function in main
    x = exp(-0.1*x)
    x = sin(6*pi*x)
    return x
#-----------------------------------

# In main
x = 2
y = f(x)
print 'f(3g)=lg' % (x, y)
```

Execution always starts with the first line in the \texttt{main}

- Nothing is computed inside a \texttt{function} before it is called

Variables initialised in the \texttt{main program} become \texttt{global variables}
Functions and branching

The main program (cont.)

1. Import functions from the math module
2. Define function f(x)
3. Define x
4. Call f and execute the function body
5. Define y as the value returned from f
6. Print a string

Lambda functions

A one-line construction of functions used to make code compact

1. \[ f = \lambda x : x^2 + 4 \]

This so-called lambda function is equivalent to the usual form

1. \[ f(x) = \begin{cases} x^2 + 4 & \text{if } x \geq 0 \\ \text{NaN} & \text{otherwise} \end{cases} \]

Lambda functions

Definition

In general, we have

1. \[ f(x_1, x_2, \ldots, x_n) = \frac{x_1^2 + x_2^2 + \cdots + x_n^2}{x_1^2 + x_2^2 + \cdots + x_n^2} \]

This can be re-written

1. \[ g = \lambda \text{arg1, arg2, arg3, \ldots} : \text{expression} \]
Lambda functions (cont.)

Lambda functions are used for function argument to functions

Example

Consider the diff2nd function used to differentiate \( g(t) = t^{-6} \) twice

- We first make a \( g(t) \) then pass \( g \) as input argument to diff2nd

We skip the step of defining \( g(t) \) and use a lambda function instead

- A lambda function \( f \) as input argument into diff2nd

```python
1d2 = diff2nd(lambda t: t**(-6), 1, h=1E-4)
```

Lambda functions (cont.)

Remark

Lambda functions can also take keyword arguments

```python
1d2 = diff2nd(lambda t, A=1, a=0.5: -a*2*t*A*exp(-a*t**2), 1.2)
```

Branching

The flow of computer programs often needs to branch

- If a condition is met, we do one thing;
- If it is not met, we do some other thing

As an example, consider the multi-case function

\[
 f(x) = \begin{cases} 
 \sin(x), & 0 \leq x \leq \pi \\
 0, & \text{elsewhere} 
\end{cases}
\]
Functions and branching

IF-ELSE blocks

**Example**

\[ f(x) = \begin{cases} 
\sin(x), & 0 \leq x \leq \pi \\
0, & \text{elsewhere} 
\end{cases} \]

Implementing this function requires a test on the value of \( x \)

Consider the following implementation

```python
def f(x):
    if 0 <= x <= pi:
        value = sin(x)
    else:
        value = 0
    return value
```

---

**IF-ELSE blocks**

**Definition**

The general structure of the **IF-ELSE** test

```python
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

- **If condition is True**, the program flow goes into the first block of statements, indented after the `if` line
- **If condition is False**, program flow goes into the second block of statements, indented after the `else:` line

The blocks of statements are indented, and note the two-points

---

**IF-ELSE blocks (cont.)**

**Example**

Consider the following code

```python
if C < -273.15:
    print 'A degree Celsius is non-physical! %d C
else:
    F = 9.0/5* C + 32 
    print F
print 'end of program'
```

We have,

- The two `print statements` in the **IF-block** are executed if and only if condition \( C < -273.15 \) evaluates as **True**
- Otherwise, execution skips the `print statements` and carries out with the computation of the statements in the **ELSE-block** and prints \( F \)
IF-ELSE blocks (cont.)

The end of program bit is printed regardless of the outcome

- This statement is not indented
- It is neither part of the IF-block nor of the ELSE-block

The computation of \( F \) will always be carried out

- The statement is not indented
- It is not part of the IF-block

This construct allows for multiple branching of the program flow
**Example**

Let us consider the so-called HAT function

\[ N(x) = \begin{cases} 
0, & x < 0 \\
 x, & 0 \leq x < 1 \\
 2 - x, & 1 \leq x < 2 \\
 0, & x \geq 2 
\end{cases} \]

Write a Python function that implements it

Consider the following implementation

```python
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    else:
        return 0
```

Consider an alternative implementation

```python
def N(x):
    if 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    else:
        return 0
```
Functions and branching

Functions
Mathematical functions as Python functions
Local and global variables
Multiple arguments
Function argument vs. global variable
Beyond math functions
Multiple return statements
No return statements
Keyword arguments
Doc strings
Functions as arguments to functions
The main program
Lambda functions
Branching
IF-ELSE blocks
Inline IF-tests

Inline IF-test

Variables are often assigned a value based on some boolean expression
Consider the following code using a common IF-ELSE test

```
1 if condition:
2     a = value1
3 else:
4     a = value2
```

The equivalent one-line syntax (inline IF-test)

```
a = (value1 if condition else value2)
```

Inline IF-test (cont.)

Remark
The IF-ELSE test cannot be used inside a lambda function

Notice that the test has more than one single expression
- Lambda functions cannot have statements
- Only a single expression is accepted

Example
Consider the following multiple-case mathematical function

\[
f(x) = \begin{cases} 
\sin(x), & 0 \leq x \leq \pi \\
0, & \text{elsewhere}
\end{cases}
\]

We are interested in the corresponding Python function

We have,

```
1 def f(x):
2     return (sin(x) if 0 <= x <= 2*pi else 0)
```

Alternatively, we have

```
1 f = lambda x: sin(x) if 0 <= x <= 2*pi else 0
```