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.

**Definition**

**Function**

A function is a collection of statements that can be run wherever and whenever needed in the program.

The function may accept input variables
- To influence what is computed inside
- (A function contains statements)

The function may return new objects

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

**Example**

Consider a function \( F(C) \) for converting degree Celsius \( C \) to Fahrenheit \( F \).

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

The function \( F \) takes \( C \) (\( C \)) 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`, followed by the function name
- Inside parentheses, a comma-separated list of function arguments
- 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
- The function output value is sent out of the function
Math functions as Python functions (cont.)

Example

The function name is \( F(F) \)

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

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

```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 \( F(C) \))

The def line (function name and arguments) is the function header

The indented statements are the function body

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

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

```
# Function header
# Function (mini) block
```

Math functions as Python functions (cont.)

Math functions as Python functions (cont.)

Example

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 in a variable

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

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

```
# Conversion function
# Conversion function
# Conversion function
# Conversion function
```

Math functions as Python functions (cont.)

Consider the usual list \( \text{Celsius} \) of temperatures in degrees Celsius

- Interest in computing a list of corresponding Fahrenheit

- We want to use function \( F \), in a list comprehension

```python
Celsius = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

- We define \( F \) for \( C \) in \( \text{Celsius} \)

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

```
# Conversion function
# Conversion function
```

Math functions as Python functions (cont.)

```python
temp1 = F(15.5)  # Given input argument 'a' (value 15.5)
print F(a+1)  # Print return value to screen (no storing)
```
Math functions as Python functions (cont.)

Example

Consider a slight variation of the F(C) function

F2(C)

We define F2(C) to return a formatted string

(Instead of a real number)

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

How to use this new function?

```
>>> s1 = F2(21)
>>> print s1
21.0 degrees Celsius correspond to 69.8 Fahrenheits
```

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, ...

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

Consider the following function

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

The (main) program around function F2(C) is not aware of variable F_value
- If invoked, an error message is returned

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

**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

**Example**

Consider the input argument to function \( F_2 \), variable \( C \)

~ Variable \( C \) is a local variable

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

# Examples to illustrate

# We cannot access variable \( C \) outside the function

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

>>> C
... #
NameError: name 'C' is not defined
```

**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**

\( C \) and \( F\_value \) are local variables

```python
# Examples to illustrate

>>> c1 = 37.5
>>> s2 = F2(c1)
>>> c1
>>> s2
... #
```

~ \( c1 \) and \( s2 \) (and \( s1 \)) are global variables
The example illustrates also that there are two different variables C.

The value of the latter (local) C is given in the call to function F3.

* When we refer to C in F3, we access the local variable.
* Inside F3, local variable C shadows global variable C.

Local variables hide/shade global variables.

~ This is important.
**Local and global variables (cont.)**

Consider now this three-line piece of code:

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

---

**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

---

**Example**

Consider the single-line piece of code

```
print sum # sum is a built-in Python function
```

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>`

---

**Value of local variable `sum` is returned, added to 1, to form an int object**

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

Final `print sum` searches global variables, it finds one (value 4)
Local and global variables (cont.)

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

Example
Consider the following piece of code

```python
a = 20; b = -2.5 # global variables

# Local and global variables

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

# Global vs Local

def f2(x):
    global a
    # a is declared global
    a = 21
    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

Multiple arguments

Functions F(C) and F2(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 (,)

```python
Functions F(C) and F2(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 (,)
```
Multiple arguments (cont.)

Example

Consider the mathematical function

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

\( g \) 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 of \( v_0 \)
- To evaluate \( y \), we need values for both \( t \) and \( v_0 \)

A natural implementation would be a function with two arguments

```python
# Example of a function with two arguments
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 are interested in the function \( y(t) = v_0 t - \frac{1}{2}gt^2 \)

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

Advantages deriving from writing `argument=value` in the call

- Reading and understanding the statement is easier.
Multiple arguments (cont.)

Suppose that the argument=value syntax is given for all arguments
- The sequence of the arguments is no longer important
- (We can place $v_0$ 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

Remark
Consider argument=value arguments

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

```
1 ####################################################################
2 def yfunc(t, v0):
3     g = 9.81
4     return v0*t - 0.5*g*t**2
5 ####################################################################
```

$\sim$ yfunc(0.1, $v_0=6$) is correct
$\sim$ yfunc($t=0.1$, 6) is illegal

Multiple arguments (cont.)

Consider the case in which $yfunc(0.1, 6)$ or $yfunc(v_0=6, t=0.1)$ is used

The arguments are automatically initialised as local variables
- The 'exist' within the function

Initialisation is the same as assigning values to variables

```
1 $t = 0.1$
2 $v_0 = 6.$
3 ####################################################################
4 def yfunc(t, v0):
5     g = 9.81
6     return v0*t - 0.5*g*t**2
7 ####################################################################
```

Such statements are not visible in the code

Multiple arguments (cont.)

$y(t) = v_0 t - \frac{1}{2} g t^2$ | $v_0 = 6$ [m s$^{-1}$], $g = 9.81$ [m s$^{-2}$]

Function argument $v$

global variable

Functions
**Functions and branching**

**UFC/DC FdP - 2019.1**

**Functions**
- Mathematical functions as Python functions
- Local and global variables
- Multiple arguments

**Function argument v global variable**

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

Mathematically, function \( y \) is understood as a function of one variable, \( t \).

A Python implementation as `function yfunc` should reflect this fact:

- `yfunc` should be a `function` of \( t \) only

Example

Consider the following construction

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

Variable \( v_0 \) is interpreted as a `global variable`.

It needs be initialised outside `function yfunc`.

- Before we attempt to call `yfunc`

```python
> v0 = 5.
> yfunc(0.6)
1.2342
```

**Function argument v global variable (cont.)**

Failing to initialise a `global variable` leads to an error message:

```
>>> yfunc(0.6)
... NameError: global name 'v0' is not defined
```

We need to define \( v_0 \) as a `global variable` prior to calling `yfunc`

```python
>>> v0 = 5.
>>> yfunc(0.6)
1.2342
```

**Beyond math functions**

- Functions as arguments to functions
- The main program
- Lambda functions
So far, Python functions have typically computed some mathematical expression, but their usefulness goes beyond mathematical functions.

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

### 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.

```python
####
# Beyond math functions (cont.)
#
# def makelist(start, stop, inc):
#  result = []
#  value = start
#  while value <= stop:
#   result.append(value)
#   value = value + inc
#  return result
#
# >>> mylist = makelist(0, 100, 0.2)
# >>> print mylist
# It will print the sequence
# 0, 0.2, 0.4, 0.6, ..., 99.8, 100
#
# Function makelist 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

Example

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

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

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

```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.

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

Multiple returns (cont.)

![Graph showing vertical position and velocity vs. time]

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

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

In the main, \( yfunc \) needs two names on LHS of the assignment operator

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

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

```
t_values = [0.05*i for i in range(10)]
for t in t_values:
    position, velocity = yfunc(t, v0=5)
    print '{:5.1g} position={:5.1g} velocity={:5.1g} %\n'.format(t, position, velocity)
```

Format \( \times 10^g \) 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
# Three objects are returned as output arguments
>>> a = f(2)
>>> a
(2.0, 4.0, 16.0)  # Stored as a tuple
```

```python
# Stored as separate variables
>>> a, x2, x4 = f(2)
```

Remark:

Functions returning multiple (comma-separated) values returns a tuple.
Suppose we are interested in creating a function to calculate the sum

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

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

Observe the terms 1/0 used to avoid integer division

\( i \) is an int object and \( x \) may also be an int

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

\( x \) and \( n \) are the input arguments

\( s \) is the return output

```python
# Summation code

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

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

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

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
Functions and branching
UFC/DC FdP - 2019.1

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

Summation (cont.)

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

The size of the terms decreases with n.

〜 The first neglected term \((n+1)\) is bigger than all remaining terms

〜 (those calculated for \(n+2, n+3, \ldots\))

Yet, it is not necessarily bigger than their sum.

The first neglected term is hence an indication of the size of the total error.

〜 We may use this term as a crude estimate of the error.

Example

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

def L2(x, n):
    s = 0
    for i in range(1, n+1):
        s += (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
```

value, approximate_error, exact_error = L2(x, 100)

No returns

Functions

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

〜 Without necessarily computing objects returned to calling code

In such situations, the return statement is not needed

〜 The function without return values

Functions and branching
UFC/DC FdP - 2019.1

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

Summation (cont.)

We return the exact error (we calculate the log function by math.log)
Consider the construction of a table of the accuracy of function $L_2(x, n)$:

$$L_2(x, n) = \sum_{k=0}^{n} \frac{x}{(k+1)x+1}$$

It is an approximation to $\ln(1 + x)$.

### Example

```python
def table(x):
    n = 1
    value_of_sum = 0
    for i in range(1, n+1):
        z = (1.0/i)(x/(1.0+1x))
        value_of_sum = z
        first_neglected_term = (1.0/(n+1))z/(1.0+1x)**(n+1)
        exact_error = log(1+x) - value_of_sum

    print(n, log(1+x), value_of_sum, first_neglected_term, exact_error)
```

For $x = 1, 2, 10, 100, 500$:

- $n=1$: $0.909091$, error: $6.22e-15$
- $n=2$: $1.32231$, error: $2.19e-01$
- $n=10$: $2.17907$, error: $2.19e-01$
- $n=100$: $2.97909$, error: $6.59e-06$
- $n=500$: $2.97909$, error: $6.22e-15$

### Notes
- Error is an order of magnitude larger than the first neglected term.
- Convergence is slower for larger values of $x$ than smaller $x$. 
No returns (cont.)

Remark

For functions w/o return statement, Python inserts an invisible one

- The invisible return is named None
- None is a special object in Python

None represents something we may think of as the ‘nothingness’

Normally, one would call function table w/o assigning return value

Yet, imagine we still assign the return value to a variable

~ The result will refer to a None object
~ result = table(500)

The None value is often used for variables that should exist in a program

- But, where it is natural to think of the value as conceptually undefined

Keyword arguments

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

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
# somefunc() can be either:
# somefunc(arg1, arg2)
# somefunc(arg1=arg1_default, arg2=arg2_default)
def somefunc(arg1=arg1_default, arg2=arg2_default):
    # arg1 and arg2 are optional arguments
    ...

# either:
# somefunc(arg1, arg2)
# somefunc(arg1=arg1_default, arg2=arg2_default)
print somefunc(arg1=arg1_default, arg2=arg2_default)
```

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
[
# def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
1
2
print arg1, arg2, kwarg1, kwarg2
3
4
#]

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

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

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

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

Remark

Keyword arguments must be listed AFTER positional arguments.

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

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,

```
import math
from math import pi, exp, sin

def f(t, A, a, omega):
    return A*exp(-a*t)*sin(omega*t)
```

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
f(2, A=3, a=1, omega=2*pi)  # We use keyword arguments
```
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

```python
>>> v2 = f(0.2, omega=1)
>>> v3 = f(0.2, omega=p, a=p, t=0.01, omega=0.1)
>>> v4 = f(0.2, 0.5, 1, 1)
```

It is natural to provide a default value for \( \epsilon \)

```python
def L3(x, epsilon=1.0E-6):
    z = float(x)
    i = 1
    term = (1.0/1.0)*(x/(1+x))**i
    while abs(term) > epsilon:
        term = (1.0/1.0)*(x/(1+x))**i
        z = z + term
    return z, i
```

Keyword arguments (cont.)

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 \( \epsilon \)

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

(Instead of specifying the number \( n \) of terms in the sum)

We make a table of the approximation error as \( \epsilon \) decreases

```python
# 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{x}{1 + x}^i

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

\[ \sim \] (Instead of specifying the number \( n \) of terms in the sum)

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 \( \epsilon \)
Keyword arguments (cont.)

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: 9.30e-10, n=187
epsilon: 1e-10, 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

Doc strings

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

Example

Consider the following Python function with plain documentation

```python
def C2F(C):
    *** Convert Celsius degrees (C) to Fahrenheit. ***
    C: Input argument, temperature in Celsius
    return: Temperature in Fahrenheit

# C: 9.0
>>> C2F(C)
```

```python
32.0
```

Doc strings (cont.)
Consider the following Python function with documentation and arguments:

```python
def line(x0, y0, x1, y1):
    a = (y1 - y0) / (x1 - x0)
    b = y0 - a * x0
    return a, b
```

**Example**

```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 \((x_0, y_0)\) and \((x_1, y_1)\).

\[
x_0, y_0: \text{a point on the line (floats)}.
\]
\[
x_1, y_1: \text{another point on the line (floats).}
\]
\[
x_0, y_0: \text{a point on the line (float objects)}.\]
\[
x_1, y_1: \text{another point on the line (float objects).}
\]
\[
return: coefficients a, b (floats) 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.

```
>>> a, b = line(1, -1, 4, 3)
1.333333333333333
-2.333333333333333
```

To extract doc strings from source code use `funcname.__doc__`
Functions (cont.)

The usual convention in Python

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

Definition

The general structure of a 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 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 $\int f(x)dx$, approximately

**Numerical solution of differential equations**

- Compute $z(t)$ from $\frac{dz}{dt} = f(x)$, approximately

In such **functions**, function $f(x)$ can be used as **input argument** ($f$)

Functions as arguments to functions (cont.)

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

<table>
<thead>
<tr>
<th>(h)</th>
<th>(g''(h))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e-10</td>
<td>42.00000</td>
</tr>
<tr>
<td>1e-09</td>
<td>41.99999</td>
</tr>
<tr>
<td>1e-08</td>
<td>42.00074</td>
</tr>
<tr>
<td>1e-07</td>
<td>42.00025</td>
</tr>
<tr>
<td>1e-06</td>
<td>42.02521</td>
</tr>
<tr>
<td>1e-05</td>
<td>47.73959</td>
</tr>
<tr>
<td>1e-04</td>
<td>-666133814</td>
</tr>
<tr>
<td>1e-03</td>
<td>-666133814.77509</td>
</tr>
<tr>
<td>1e-02</td>
<td>-666133814.77509</td>
</tr>
<tr>
<td>1e-01</td>
<td>-666133814.77509</td>
</tr>
</tbody>
</table>

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

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

- For small \(h\) rounding errors blow up and destroy accuracy
- It is necessary to switch from standard floating-point numbers (\texttt{float}) to numbers with arbitrary high precision (module \texttt{decimal})
The main program

Example

```python
from math import *  # In main

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

# In main
x = 2
y = f(x)
print f(5g)+%g (x, y)
```

Execution always starts with the first line in the main

- When a function is encountered, its statements are used to define it
  - Nothing is computed inside a function before it is called

Variables initialised in the main program become global variables
Lambda functions

In general, we have the following

\begin{align*}
def g(\text{arg1}, \text{arg2}, \text{arg3}, \ldots): & \\
& \text{return expression} \\
\end{align*}

This can be re-written

\begin{align*}
g = \text{lambda arg1, arg2, arg3, \ldots: expression} \\
\end{align*}

Lambda functions (cont.)

Example

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

- We first make a \( g(t) \) then pass \( g \) as \texttt{input} argument to \texttt{diff2nd}
Lambda functions (cont.)

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

```python
>>> d2 = diff2nd(lambda t: t**(-6), 1.0, h=1E-4)
```

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

Lambda functions (cont.)

**Remark:**

Lambda functions can also take keyword arguments:

```python
d2 = diff2nd(lambda t, A=1, a=0.5: -a*2*t*A*exp(-a*t**2), 1.0)
```

Branching

Functions and branching

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
Branching

Example
Consider the multi-case function
\[ f(x) = \begin{cases} \sin(x), & 0 \leq x \leq \pi \\ 0, & \text{elsewhere} \end{cases} \]

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 \( \text{condition is True} \), the program flow goes into the first block of statements, indented after the `if` line.
- If \( \text{condition is False} \), program flow goes into the second block of statements, indented after the `else` line.

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

IF-ELSE blocks (cont.)

Example
Consider the following code:

```python
if C < -273.15:
    print '5 degree Celsius is non-physical!' % 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 \( F \).
Functions and branching

IF-ELSE blocks (cont.)

Consider the following code:

1. if C < -273.15:
2.   print '16 degrees Celsius is non-physical!' % C
3. else:
4.   F = 9.0/5*C + 32
5.   print F
6. end of program

The end of program bit is printed regardless of the condition check outcome.

~ This statement is not indented

It is neither part of the IF-block nor of the ELSE-block.

Definition

The else part of the IF-ELSE test can be skipped.

IF-ELSE blocks (cont.)

Example

Consider the following code:

1. if C < -273.15:
2.   print '16 degrees Celsius is non-physical!' % C
3.   F = 9.0/5*C + 32
4. else:
5.   print F
6. end of program

The computation of F will always be carried out:

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

Definition

With else (for else if) several mutually exclusive IF-test are performed.

This construct allows for multiple branching of the program flow.
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 0.0
```

Consider an alternative implementation

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

### IF-ELSE blocks (cont.)

#### Example

Let us consider the so-called HAT function

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

Write a Python function that implements it.
Inline IF-test

Variables are often assigned a value based on some boolean expression.

Consider the following code using a common **IF-ELSE test**

```python
if condition:
a = value1
else:
a = value2
```

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

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

**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,

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

Alternatively, we have

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

**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.