Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 3
Using and Designing Functions

What are functions?, Python Built-in Functions, Local Variables, Designing Functions

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

Abstract

"No amount of genius can overcome obsession with detail."
– Traditional

We describe the general concept of functions and functions in Python specifically.
We will learn about some functions provided by Python and how to design our own functions.
We will introduce a powerful new concept: local variables.
We will write our first program by combining functions.

Overview

- You probably know functions from mathematics
- Functions in Python are similar
- Python provides many useful functions
- We define our own functions to group expressions and give them a name
- This way, we can reuse the expressions

Functions are important building blocks of programs.

Functions Provided by Python

- Python provides many useful *built-in* functions
- For example, `abs` produces the absolute value of a number:

```
>>> abs(-3)
3
>>> abs(4.2)
4.2
```

- The above statements are *function calls*
- The general form of a function call is:

```
function_name([argument, ...])
```

We use functions by *calling* them with *arguments*.

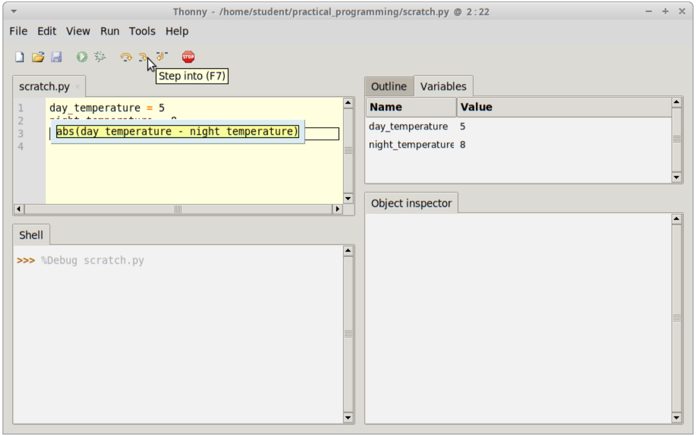
Function Arguments

- Arguments are expressions that appear in the parentheses of a function call
- In particular, the argument expression may contain variables:

```
>>> day_temperature = 5
>>> night_temperature = 8
>>> abs(day_temperature - night_temperature)
3
```

Arguments are the inputs to a function.

Try this in the IDE Debugger (Examination Mode)



Carefully observe the evaluation order.

Rules to executing a function call

1. Evaluate each argument expression, from left to right.
2. Pass the resulting values into the function.
3. Execute the function.

The function call produces a value.

Function Calls in Arguments

- Functions produce values
- This means they can appear in expressions:

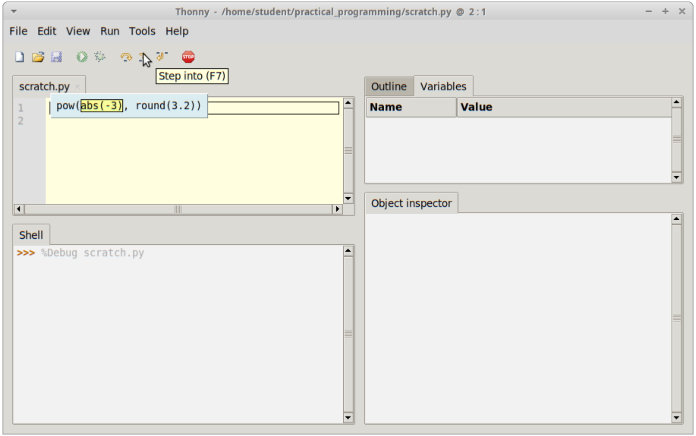
```
>>> abs(-5) + abs(1.4)
>>> 6.4
```

- Therefore we can use function calls as arguments:

```
>>> pow(abs(-3), round(3.2))
27
```

Nesting function calls is a very common technique in programming.

Try this in the IDE Debugger (Examination Mode)



Carefully observe the evaluation order.

Built-in Functions: Type Conversions

- Converting from one type to another is very useful
- For example, `int` and `float` can be used as functions:

```
>>> int(7.81)
7
>>> int(-5.2)
-5
>>> float(19)
19.0
```

Note that calling `int` does truncate, not round.

Asking for Help

- If you are unsure, ask Python for help
- You can do this by calling the `help` function with an object or type as an argument:

```
>>> help(pow)
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

The `help` function works for all built-ins. We will also make it work for our own objects.

Why We Want Your Own Functions

- The built-in functions are useful but have to be generic
- We often want functions that help us solve our specific problems
- For example, it would be nice to do this:

```
>>> convert_to_celsius(92.3)
33.5
>>> convert_to_celsius(13.1)
-10.5
```

- Python is not psychic, though:

```
>>> convert_to_celsius(92.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'convert_to_celsius' is not defined
```

With functions we don’t have to repeat the same expression.

Defining Your Own Functions

- The general form of a *function definition* is:

```
def function_name([parameter, ...]):
    block
```

- The block *must* contain at least one statement
- It usually contains one or more **`return`** statements:

```
return expression
```

- For example:

```
def convert_to_celsius(fahrenheit):
    return (fahrenheit - 32) * (5 / 9)
```

Function definitions are Python statements that create *function objects*.

Defining & Calling Functions

1. Python executes the function definition, creating a function object
2. Next, the function call `convert_to_celsius(102)` is executed, this assigns `102` to the parameter `fahrenheit`
3. Now the return statement is executed, this involves evaluating the returned expression `(fahrenheit - 32) * (5 / 9)`
4. When the function call is completed, Python continues with the statement after the call

```
1  def convert_to_celsius(fahrenheit):
3      return (fahrenheit - 32) * (5 / 9)

2  convert_to_celsius(102)

4  # rest of program...
```

Follow this in the IDE debugger.

Words that are special to Python

- Some words are special to Python
- We can't use them except as Python intends
- This is the full list:

```
False    break     else     if        not     while
None     class     except   import    or      with
True     continue  finally  in        pass    yield
and      def       for      is        raise
as       del       from     lambda    return
assert   elif      global   nonlocal  try
```

The special words are called *keywords*. You can't redefine them.

Temporay Storage: Local Variables

- It is a good idea to break down complex computations
- This requires temporary storage, or *local variables*
- For example:

```
def quadratic(a, b, c, x):
    quadratic_term = a * (x ** 2)
    linear_term = b * x
    constant_term = c
    return quadratic_term + linear_term + constant_term
```

Python creates a local variable when an expression is assigned to it.

Temporay Storage: Local Variables

- Local variables can’t be used outside of the function:

```
>>> quadratic(3, 4, 1, 4.1)
67.82999999999998
>>> quadratic_term
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'quadratic_term' is not defined
```

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'a' is not defined
```

Parameters are also local variables.

Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

- Can you predict what this code does?
- It is a bit confusing because of the multiple use of `x`
- You have to understand local variables
- A local variable is local to a *namespace*
- Python creates a namespace when executing a function call

When we call a function:

1. Evaluate the arguments left to right
2. Create a namespace to hold local variables
3. Assign the argument values to the parameters
4. Execute the function body

You can think of namespaces as different rooms.

Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

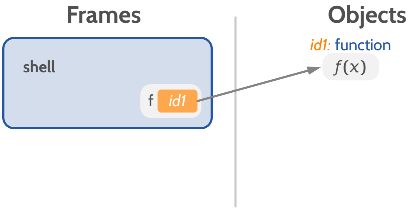
- From now on, we will reflect namespaces in our memory model diagrams
- We will draw separate boxes for different areas of computer memory
- Programmers call these boxes *frames*

Frames	Objects
Frames for namespaces go here	*Objects go here*

You can think of frames as pieces of scratch paper.

Detailed Function Definition & Call Example

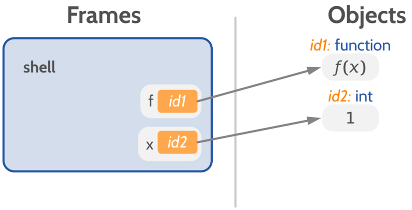
```
► >>> def f(x):
  ...     x *= 2
  ...     return x
  ...
  >>> x = 1
  >>> x = f(x + 1) + f(x + 2)
```



- Python is about to execute the function definition
- We indicate the current line of the code with a marker on the left
- When Python executes the function definition it creates a function object and assigns its address (`id1`) to the variable `f` in the shell's frame

Detailed Function Definition & Call Example

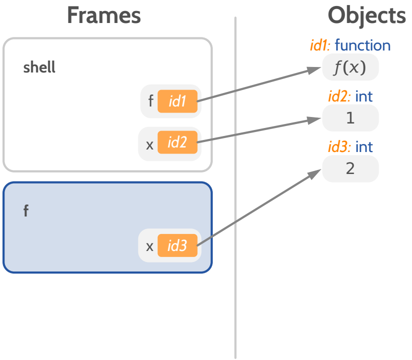
```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```



- Now Python executes the first assignment in the shell
- This adds the variable x to the shell's frame
- The variable x in the shell's frame now refers to the object of type `int` at address `id2` with the value 1

Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

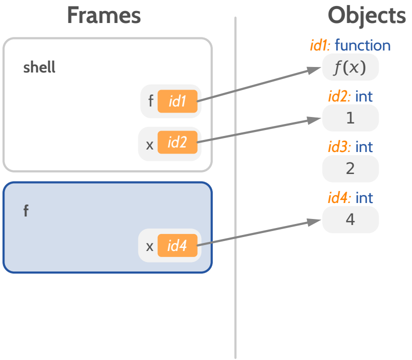


- Next the second assignment in the shell is executed
- This involves evaluating the expression on the right hand side of the assignment
- First, the expression `x + 1` is evaluated
- The value of `x` in the shell's frame is `1`
- The expression evaluates to `2`
- The value `2` is assigned to the parameter `x`

Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

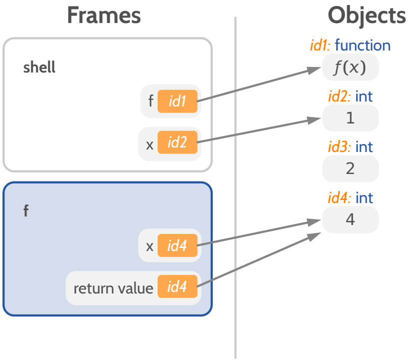
- Now Python executes the function body
- The local variable x is doubled via an augmented assignment operator
- The local variable x now refers to the object at address `id4` which is of type `int` and has the value 4
- No variable refers to the object at `id3` anymore



Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

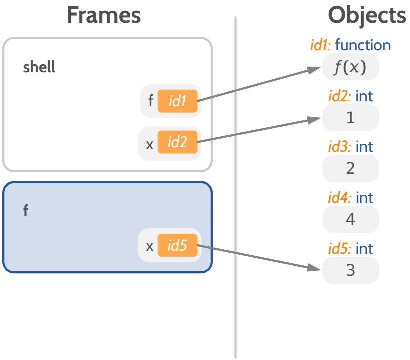
- Next, the `return` statement is executed
- This finishes the first function call
- The first part of the assignment expression is evaluated
- It results in the object of type `int` with the value 4 at address `id4`



Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
► >>> x = f(x + 1) + f(x + 2)
```

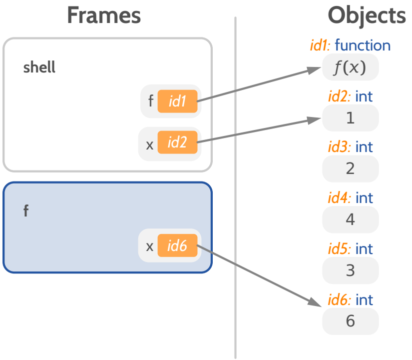
- We are now back to evaluating the expression `x + 2` in the shell
- This yields an object of type `int` of value `3` at the memory address `id5`
- The function parameter `x` now refers to the object at address `id5`



Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

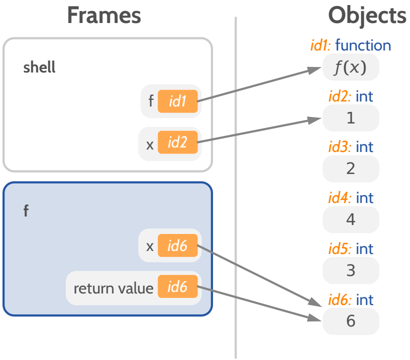
- Python executes the first statement in the function
- The local variable x is doubled via an augmented assignment operator
- The local variable x now refers to the object at address `id6` which is of type `int` and has the value 6



Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

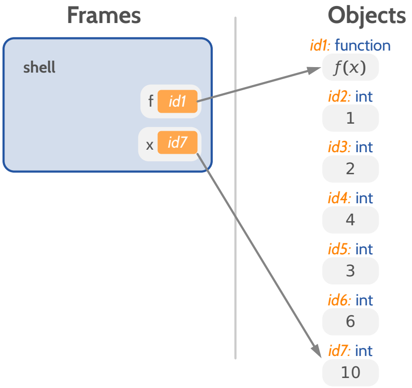
- The `return` statement for the second function call is executed
- The second part of the assignment expression is evaluated
- It results in the object of type `int` with the value 6 at address `id6`



Detailed Function Definition & Call Example

```
>>> def f(x):
...     x *= 2
...     return x
...
>>> x = 1
► >>> x = f(x + 1) + f(x + 2)
```

- The right hand side of the assignment is fully evaluated
- It results in an object of type `int` with the value `10` at address `id7`
- The variable `x` in the shell's frame now refers to the object at address `id7`



Python does all this for you. As a good programmer you need to know these details.

The Memory Model: Object Identities

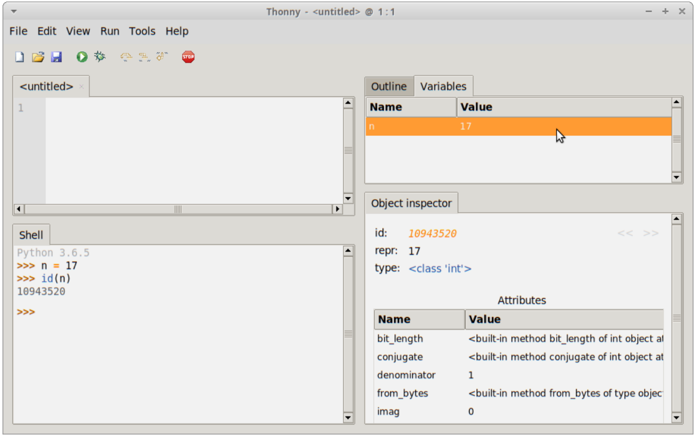
```
>>> n = 17
>>> id(n)
10919936
```

```
>>> help(id)
Help on built-in function id in module builtins:

id(obj, /)
Return the identity of an object.

This is guaranteed to be unique among simultaneously existing objects.
(CPython uses the object's memory address.)
```

- You can use the built-in function `id` to find an object's identity
- This is not just cool but very helpful
- Note that objects can be *equivalent* while *not* being identical

Identities are unique. Their meaning under the hood depends on the Python implementation.

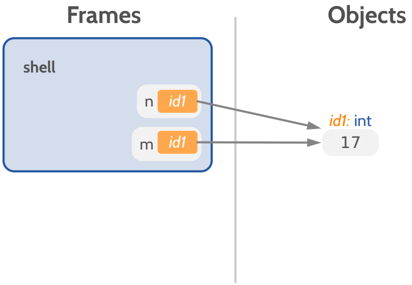
Try this in the IDE Shell



Familiarize yourself with the object inspector.

Aliasing & Caching

```
>>> n = 17
>>> m = n
>>> k = 17
```

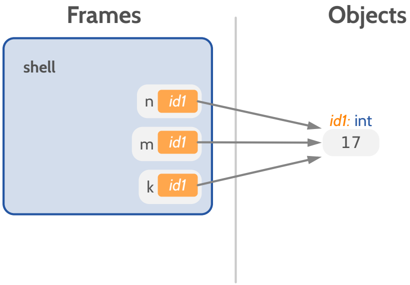


- Several variables can refer to the same object
- This is called *aliasing*

Aliasing will become more interesting with mutable objects.

Aliasing & Caching

```
  >>> n = 17
  >>> m = n
► >>> k = 17
```



- Keeping objects around in case they might be used is called *caching*
- Python automatically caches small objects
- This is notable for small integers

You don't have to worry about memory management in Python.

The Structure of Good Functions

- Writing a good function requires planning
- What is the name of the function?
- What are the parameters?
- What does the function return?
- The function must be *documented* well
- Examples are an important part of the documentation

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

Good naming and documentation are very important.

The Structure of Good Functions

- The first line is the function *header*

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the order of appearance, not the order you should think about things.

The Structure of Good Functions

- The first line is the function *header*
- This is followed by the *docstring*

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the order of appearance, not the order you should think about things.

The Structure of Good Functions

- The first line is the function *header*
- This is followed by the *docstring*
  - One line summary

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the order of appearance, not the order you should think about things.

The Structure of Good Functions

- The first line is the function *header*
- This is followed by the *docstring*
  - One line summary
  - Detailed description

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the order of appearance, not the order you should think about things.

The Structure of Good Functions

- The first line is the function *header*
- This is followed by the *docstring*
    - One line summary
    - Detailed description
    - Examples

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the order of appearance, not the order you should think about things.

The Structure of Good Functions

- The first line is the function *header*
- This is followed by the *docstring*
    - One line summary
    - Detailed description
    - Examples
- Function *body*

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the order of appearance, not the order you should think about things.

Designing Your Own Functions

- Think of the examples first
- What are the function parameters?
- What *exactly* should the function do?
- The examples should cover *edge cases*
- For example, what happens when the two days are the same?

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the function design recipe we recommend to follow.

Designing Your Own Functions

- Next, think of the short description
- If you can't come up with one, this indicates a problem
- It should fit on one line
- Make it *prescriptive*, not *descriptive*

```
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the function design recipe we recommend to follow.

Designing Your Own Functions

- Now it's time to write the function header
- The name should clearly convey what the function does
- Pick meaningful parameter names
- Make it easy for other programmers to use your function

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the function design recipe we recommend to follow.

Designing Your Own Functions

- Now write the function *description*
- This should be a short paragraph describing what the function does
- Make it clear to other programmers what the inputs and the return value are
- If it is useful for users of your function you can also briefly mention *how* the function works
- You can omit this for very simple functions

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the function design recipe we recommend to follow.

Designing Your Own Functions

- Finally, write the function body
- The body should be reasonably short
- If it has many lines think about a way to break it up
- That said, functions sometimes need to be a bit lengthy

```
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

This is the function design recipe we recommend to follow.

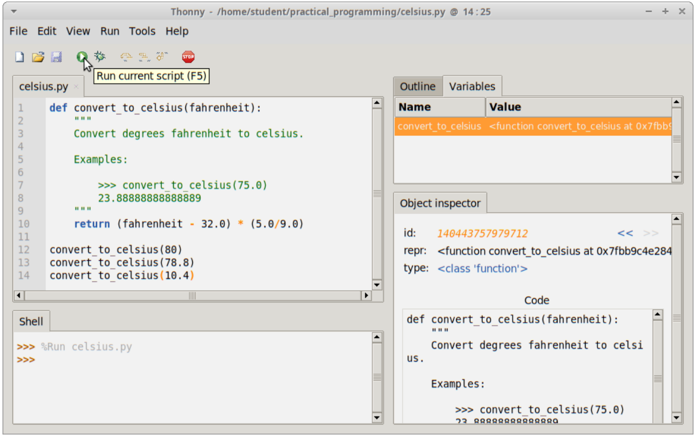
Designing Your Own Functions

- Now test your function
- Try all examples
- If one does not work, something is wrong with the function body
- Then you might need more tests to figure out what is wrong

```python
def days_difference(day1, day2):
    """
    Return the number of days between day1 and day2.

    The two days are assumed to be in
    the range 1-365, that is they
    indicate a day of the year.

    Examples:

        >>> days_difference(200, 224)
        24
        >>> days_difference(47, 47)
        0
        >>> days_difference(100, 99)
        -1
    """
    return day2 - day1
```

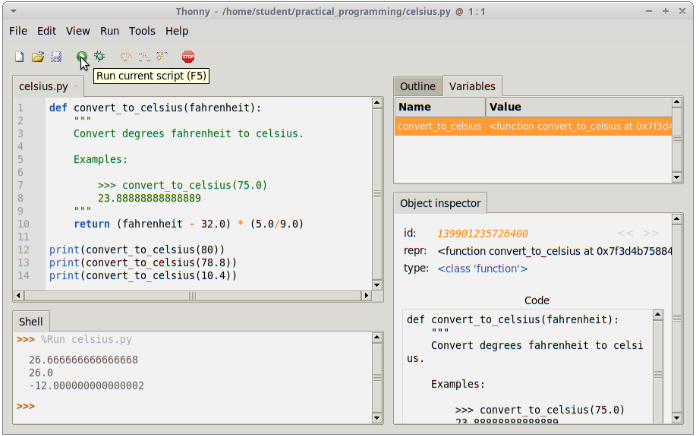
This is the function design recipe we recommend to follow.

Running a Program in the IDE



There seem to be no results.

Running a Program in the IDE



We used the `print` function to show results. We will learn other ways to run programs later.

Functions That Don’t Return a Value

- You can write a function without a **`return`** statement
- How can this possibly be useful?
- In fact it is a bad sign
- But it can be useful or necessary
- You need to know what happens

```
>>> def f(x):
...     x *= 2
...
>>> res = f(3)
>>> res
```

Why does this not cause an error?

Functions That Don’t Return a Value

- There is no error because *all* functions in Python return a value
- If you do not write a **`return`** statement **`None`** is returned
- You can also explicitly return **`None`**
- It makes no difference

```
>>> def f(x):
...     x *= 2
...
>>> res = f(3)
>>> print(res)
None
>>> id(res)
10748000
```

```
>>> def f(x):
...     x *= 2
...     return None
...
>>> res = f(3)
>>> print(res)
None
>>> id(res)
10748000
```

We will learn more about the `None` object later. It is very useful.

Exercises Lecture 3