CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

PRG – PROGRAMMING ESSENTIALS

Lecture 3 – Program structure, Functions

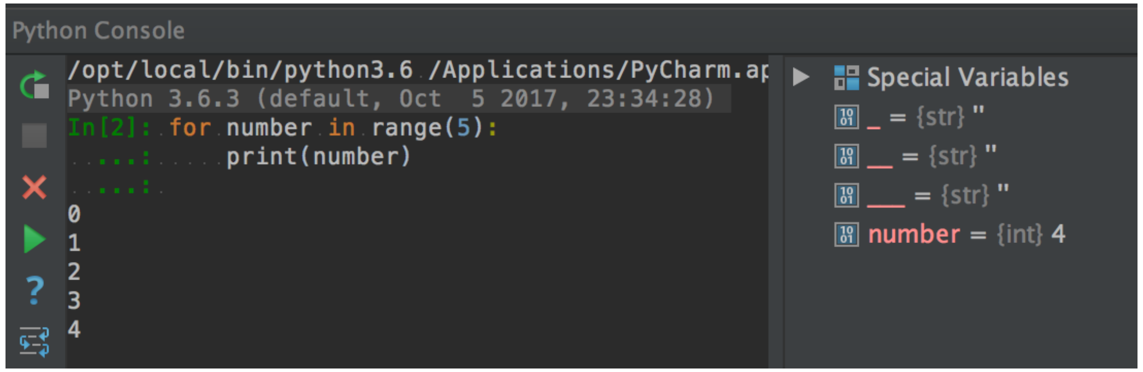
Milan Nemy

**Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics**

https://beat.ciirc.cvut.cz/people/milan-nemy/

milan.nemy@cvut.cz

RECAP: LOOPS – FOR, WHILE



```
/opt/local/bin/python3.6 /Applications/PyCharm.ap
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: for number in range(5):
   ...:     print(number)
   ...: 
0
1
2
3
4
```

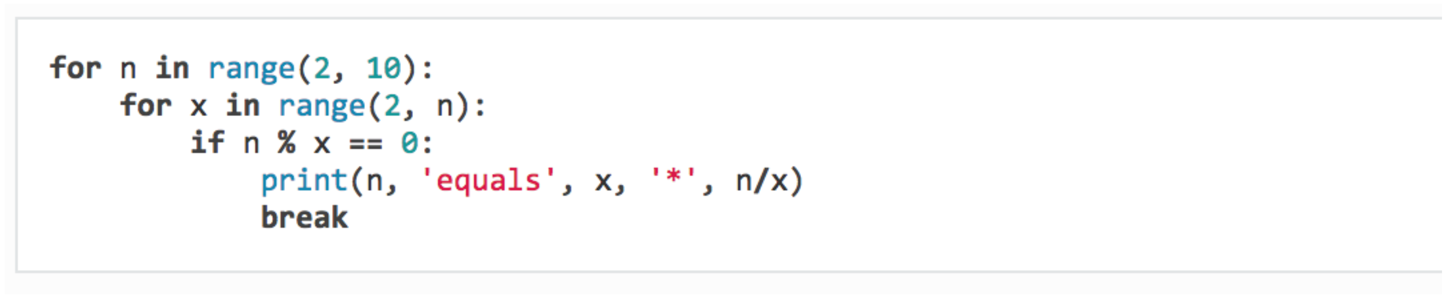
```
_ = {str} ''
__ = {str} ''
___ = {str} ''
number = {int} 4
```

<ins>On each iteration or pass of the loop</ins>:

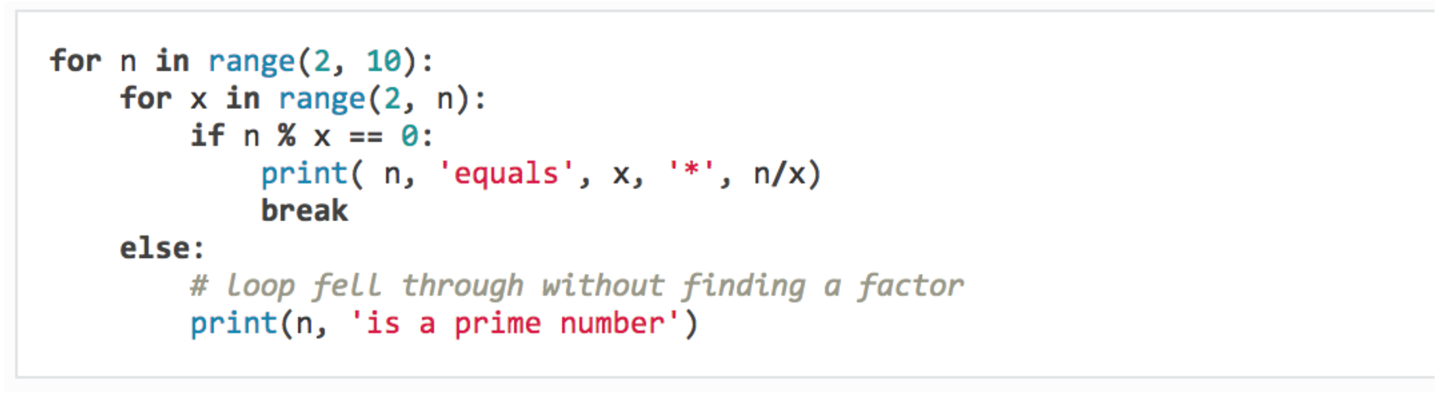
- Check to see if there are still more **items to be processed**
- If there are **none** left (the **terminating condition** of the loop) the loop has finished
- If there are items still to be processed, the **loop variable is updated** to refer to the next item in the list
- Program **continues at the next statement** after the loop body
- To explore: early **break**, or **for – else** loop, **while loop**

source http://openbookproject.net/thinkcs/python/english3e/hello_little_turtles.html

RECAP: LOOPS & CONDITIONS



```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
```



```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n/x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

- Recommendation: **early return** / **early break**
- Special condition: **FOR – ELSE**
- Explore on your own: **for, in, while, if, else, break, continue**

source http://book.pythontips.com/en/latest/for_-_else.html

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

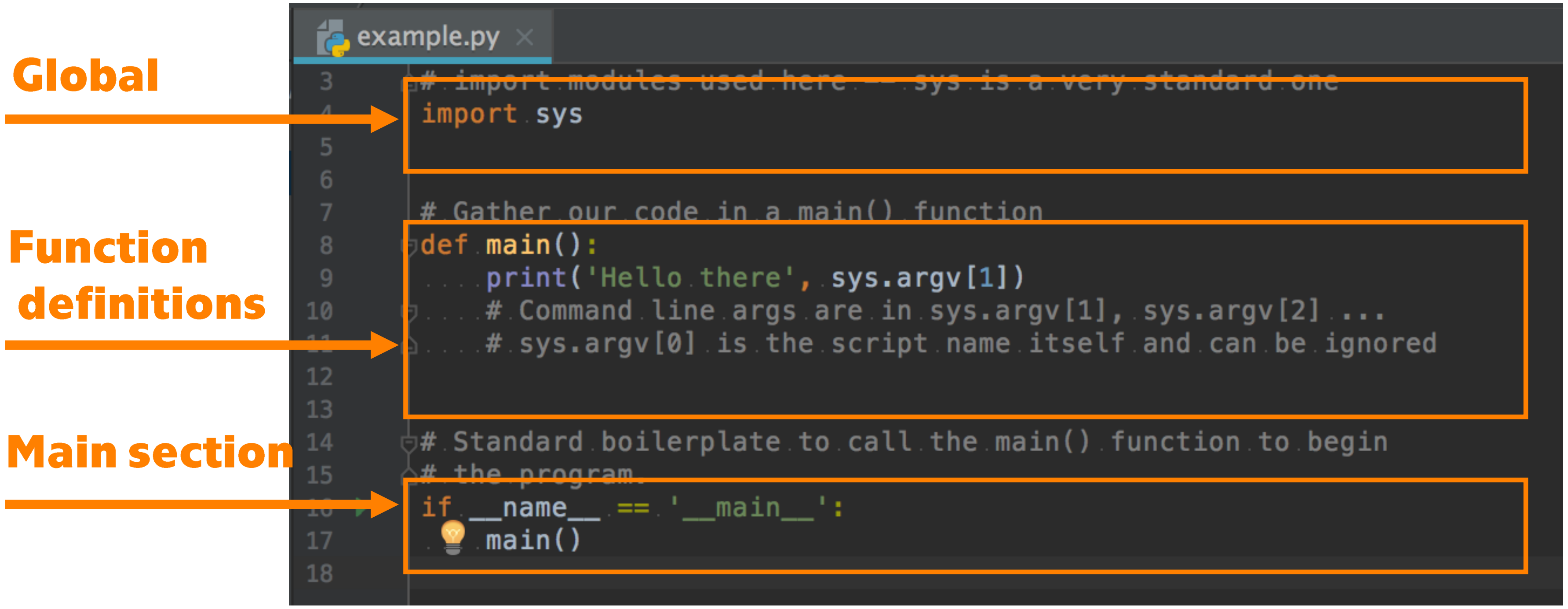
HIERARCHY OF TYPES

Type	Example
object	
- str	`“look”`
- int	`3`
- bool	`True`
- float	`3.0`
- NoneType	`None`

- Data types have hierarchy
- “Everything is an object”

source http://book.pythontips.com/en/latest/for_-_else.html

PROGRAM STRUCTURE

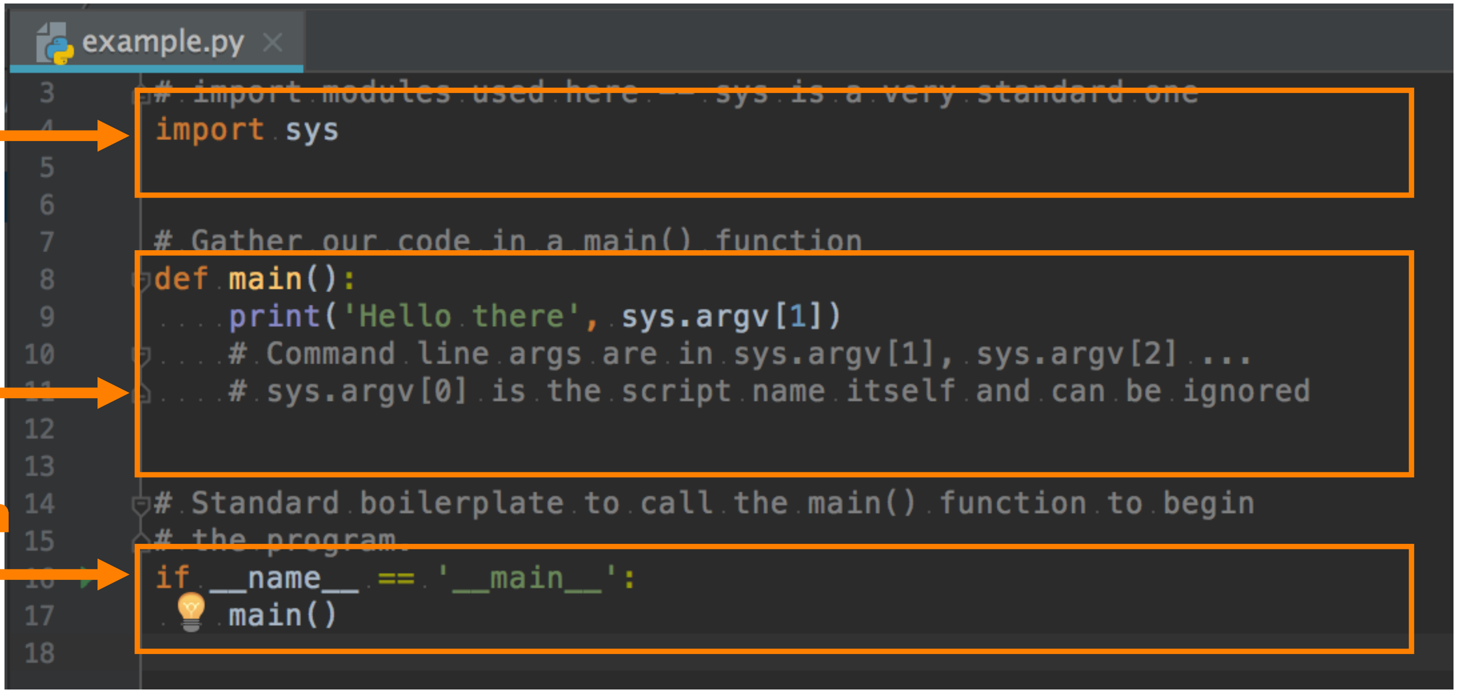


```
# import modules used here -- sys is a very standard one
import sys


# Gather our code in a main() function
def main():
    print('Hello there', sys.argv[1])
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored


# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()

```

1. **Global** definitions section
2. **Function definitions / classes definitions** section
3. **Sequence of instructions** section (here the main section)

source https://developers.google.com/edu/python/introduction

PROGRAM STRUCTURE

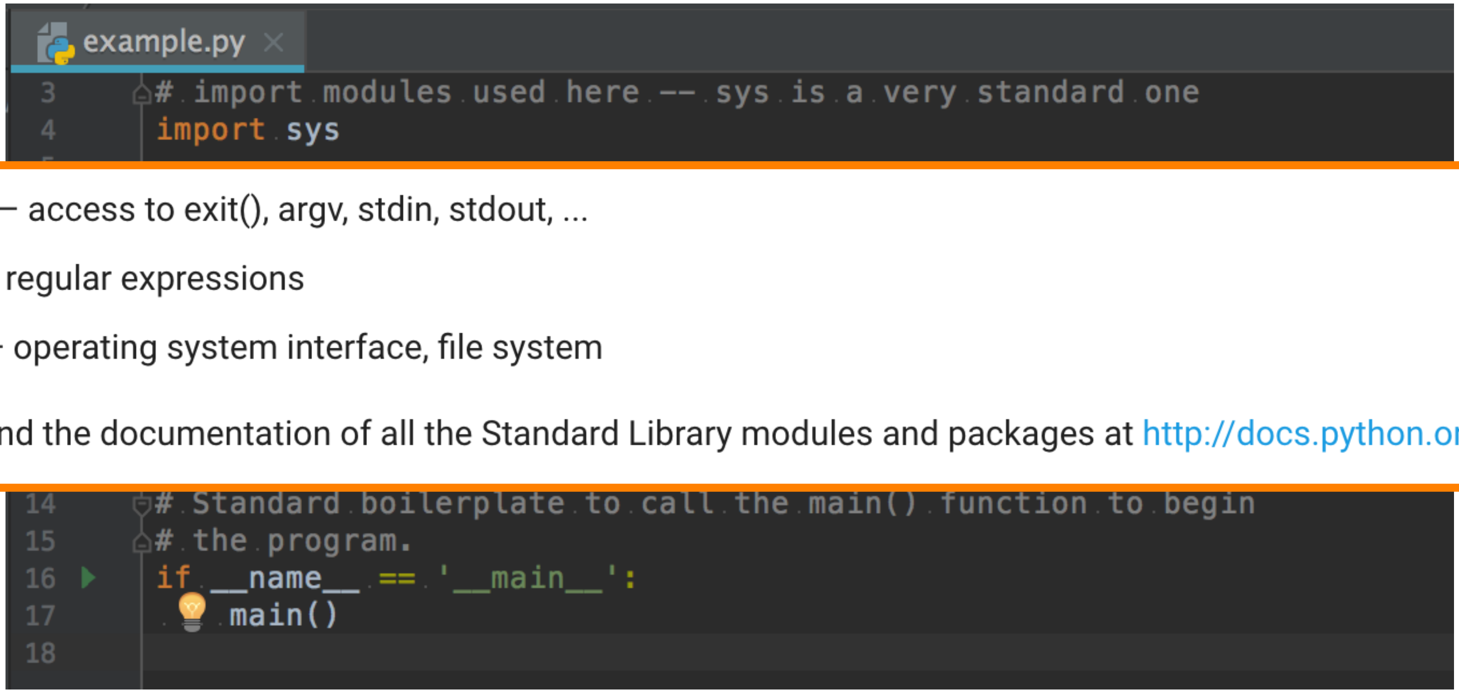


```python
# import modules used here -- sys is a very standard one
import sys


# Gather our code in a main() function
def main():
    print('Hello there', sys.argv[1])
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored


# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

- When python interpreter runs a source file as main program, it sets **__name__** variable to have a value **"__main__"**
- If being imported from another module, **__name__** will be set to the **module's name**

source https://developers.google.com/edu/python/introduction

PROGRAM STRUCTURE



```
# import modules used here -- sys is a very standard one
import sys
```

- sys — access to exit(), argv, stdin, stdout, ...
- re — regular expressions
- os — operating system interface, file system

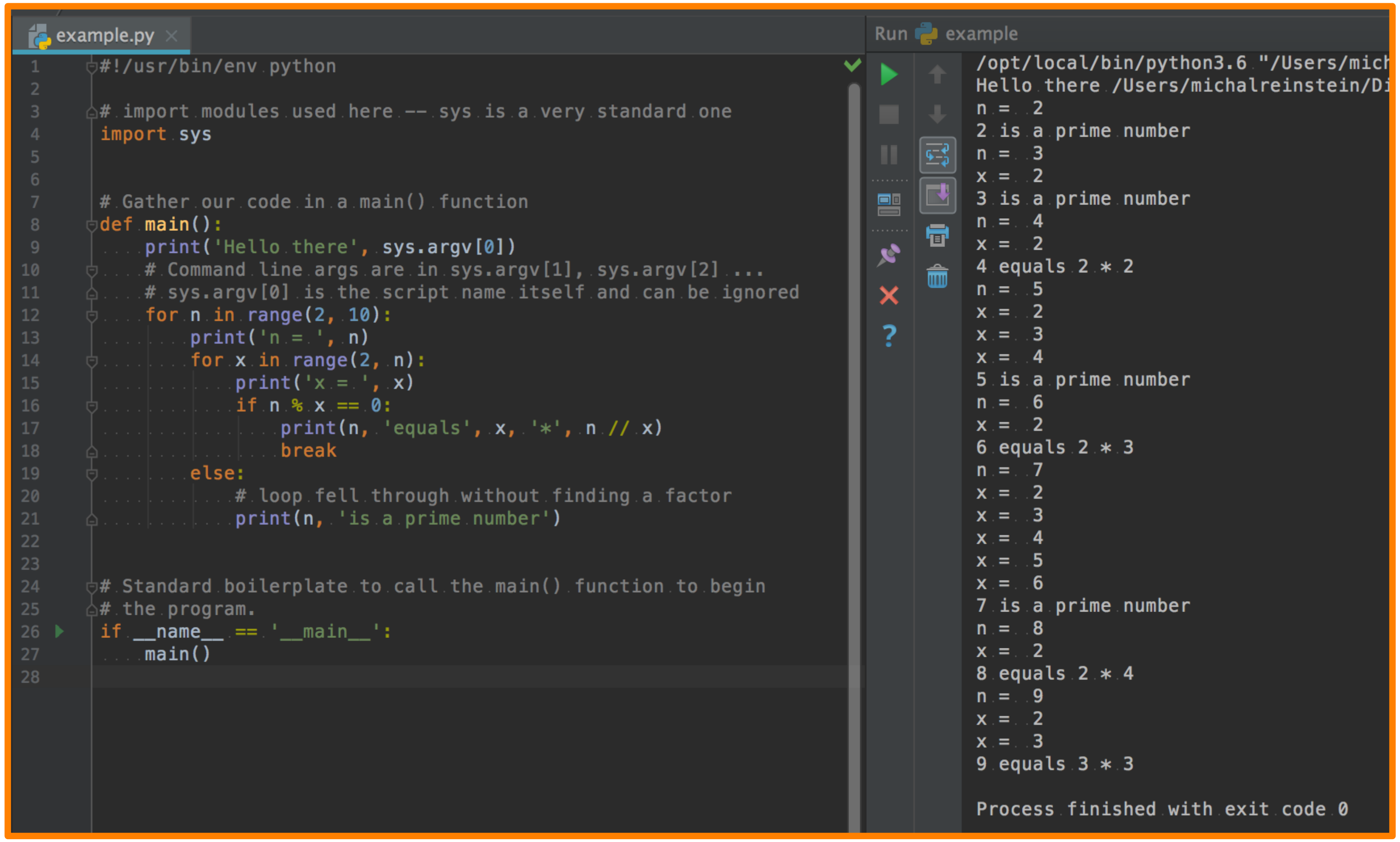
You can find the documentation of all the Standard Library modules and packages at http://docs.python.org/library.

```
# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

- Use **import** to include functions / classes from other modules

source https://developers.google.com/edu/python/introduction

EXAMPLE



```python
#!/usr/bin/env python

# import modules used here -- sys is a very standard one
import sys


# Gather our code in a main() function
def main():
    print('Hello there', sys.argv[0])
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored
    for n in range(2, 10):
        print('n = ', n)
        for x in range(2, n):
            print('x = ', x)
            if n % x == 0:
                print(n, 'equals', x, '*', n // x)
                break
        else:
            # loop fell through without finding a factor
            print(n, 'is a prime number')


# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

```
/opt/local/bin/python3.6 "/Users/mich
Hello there /Users/michalreinstein/Di
n =  2
2 is a prime number
n =  3
x =  2
3 is a prime number
n =  4
x =  2
4 equals 2 * 2
n =  5
x =  2
x =  3
x =  4
5 is a prime number
n =  6
x =  2
6 equals 2 * 3
n =  7
x =  2
x =  3
x =  4
x =  5
x =  6
7 is a prime number
n =  8
x =  2
8 equals 2 * 4
n =  9
x =  2
x =  3
9 equals 3 * 3

Process finished with exit code 0
```

WHY FUNCTIONS?

- Organize program into **chunks** that match **how we think** about the problem
- Code **re-using** without copy-paste
- Enforcing logical **structure** into the code
- Easier **debugging**
- Code **readability**

source http://openbookproject.net/thinkcs/python/english3e/functions.html

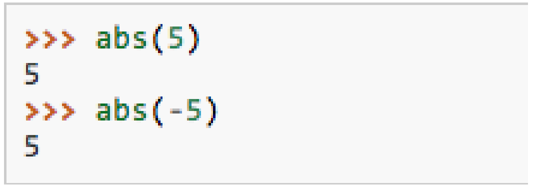
FUNCTION DEFINITION

```
def NAME( PARAMETERS ):
    STATEMENTS
```

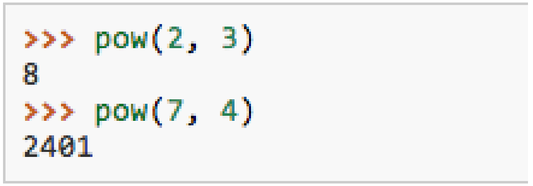
- **Function** = named sequence of statements belonging together
- **Header line**: begins with a keyword **def**, ends with a colon **:**
- **Body**: one or more statements, each indented the same amount
- **Parameter list**: empty or any number of comma separated parameters (can have default value)
- Any **name** except for keywords and illegal identifiers
- Any **number of statements** inside the function, but **indented** from the **def** (standard indentation of **four spaces**)
- Function may or may not produce a result

source http://openbookproject.net/thinkcs/python/english3e/functions.html

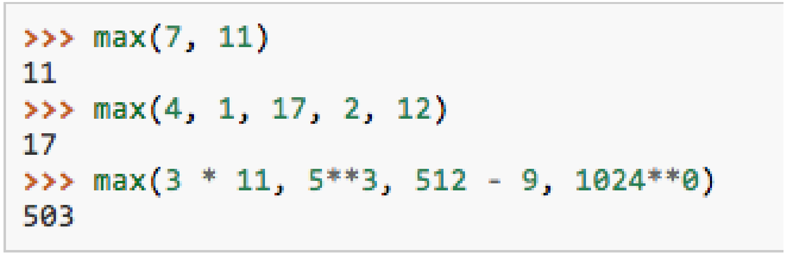
FUNCTIONS WITH ARGUMENTS



```
>>> abs(5)
5
>>> abs(-5)
5
```



```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```



```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5**3, 512 - 9, 1024**0)
503
```

- Most functions require **arguments** (named arguments, default values)
- More than one argument: e.g. **pow(base, exponent)**
- Functions like **range**, **int**, **abs** all return values that can be used to build more complex expressions
- Function that returns value is called a **fruitful function**
- Opposite of a fruitful function is **void function** (procedure)

source http://openbookproject.net/thinkcs/python/english3e/functions.html

LOCAL VARIABLES

```python
def final_amt(p, r, n, t):
    a = p * (1 + r/n) ** (n*t)
    return a
```

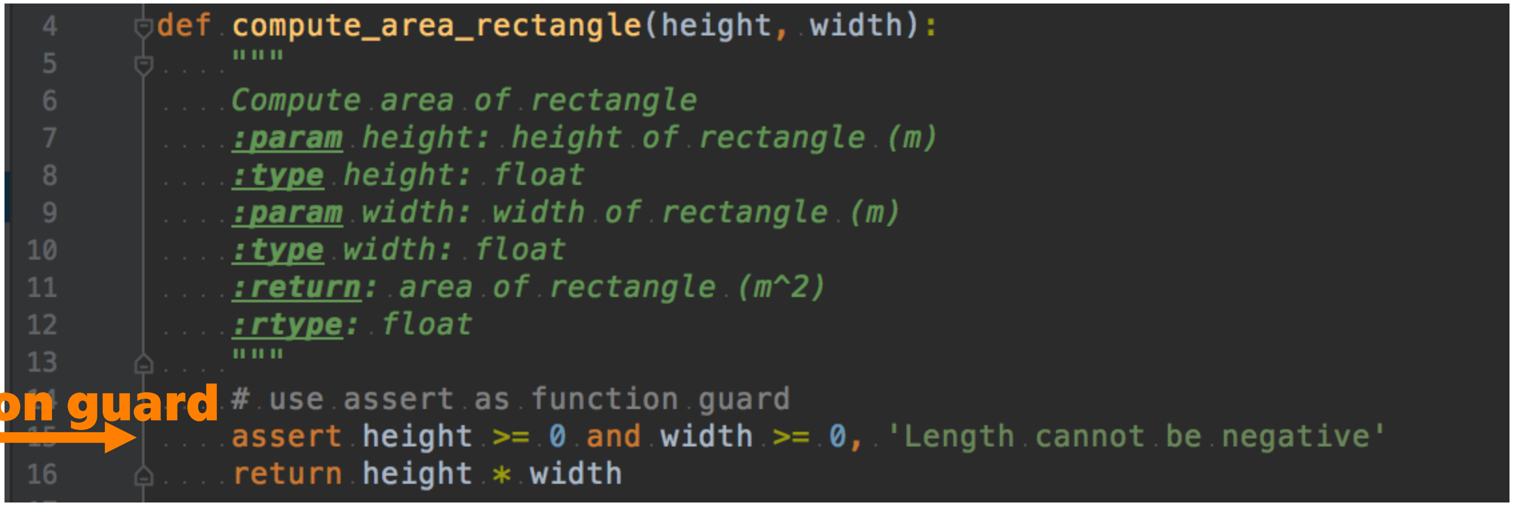
If we try to use `a`, outside the function, we'll get an error:

```
>>> a
NameError: name 'a' is not defined
```

- When a variable is created inside a function, it is **local** and cannot be used outside (**shadowing names**)
- The variable **a** is local to **final_amt**
- Local variables only exist while the function is being executed — this is called variable **lifetime**
- Parameters are local and act like local variables

source http://openbookproject.net/thinkcs/python/english3e/functions.html

DOCSTRINGS



```
def compute_area_rectangle(height, width):
    """
    Compute area of rectangle
    :param height: height of rectangle (m)
    :type height: float
    :param width: width of rectangle (m)
    :type width: float
    :return: area of rectangle (m^2)
    :rtype: float
    """
    # use assert as function guard
    assert height >= 0 and width >= 0, 'Length cannot be negative'
    return height * width
```

Function guard

- Docstrings are meant for **documentation** (if the first thing after the function header is string then treated as docstring)
- Key way to **document** our functions
- Concept of abstraction (need to know the interface)
- Formed using **triple-quoted** strings
- Different from comments: retrievable by Python tools at **runtime** (comments are completely eliminated during parsing)

source http://openbookproject.net/thinkcs/python/english3e/functions.html

FRUITFUL FUNCTIONS

Temporary variable

```python
def area(radius):
    b = 3.14159 * radius**2
    return b
```

```python
def area(radius):
    return 3.14159 * radius * radius
```

- Functions such as **abs**, **pow**, **int**, **max**, **range**, produce results
- Return statement of fruitful functions includes a **return value**
- Temporary variables like **b** above make **debugging** easier

source http://openbookproject.net/thinkcs/python/english3e/fruitful_functions.html

FRUITFUL FUNCTIONS

```python
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

```python
def absolute_value(x):
    if x < 0:
        return -x
    return x
```

```python
def bad_absolute_value(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

```
>>> print(bad_absolute_value(0))
None
```

- Multiple return statements, one in each branch of conditional
- Code after return is called **dead code**, or unreachable code
- All Python functions return **None** whenever they do not return another value.

source http://openbookproject.net/thinkcs/python/english3e/fruitful_functions.html

BOOLEAN FUNCTIONS

```python
def is_divisible(x, y):
    """ Test if x is exactly divisible by y """
    if x % y == 0:
        return True
    else:
        return False
```

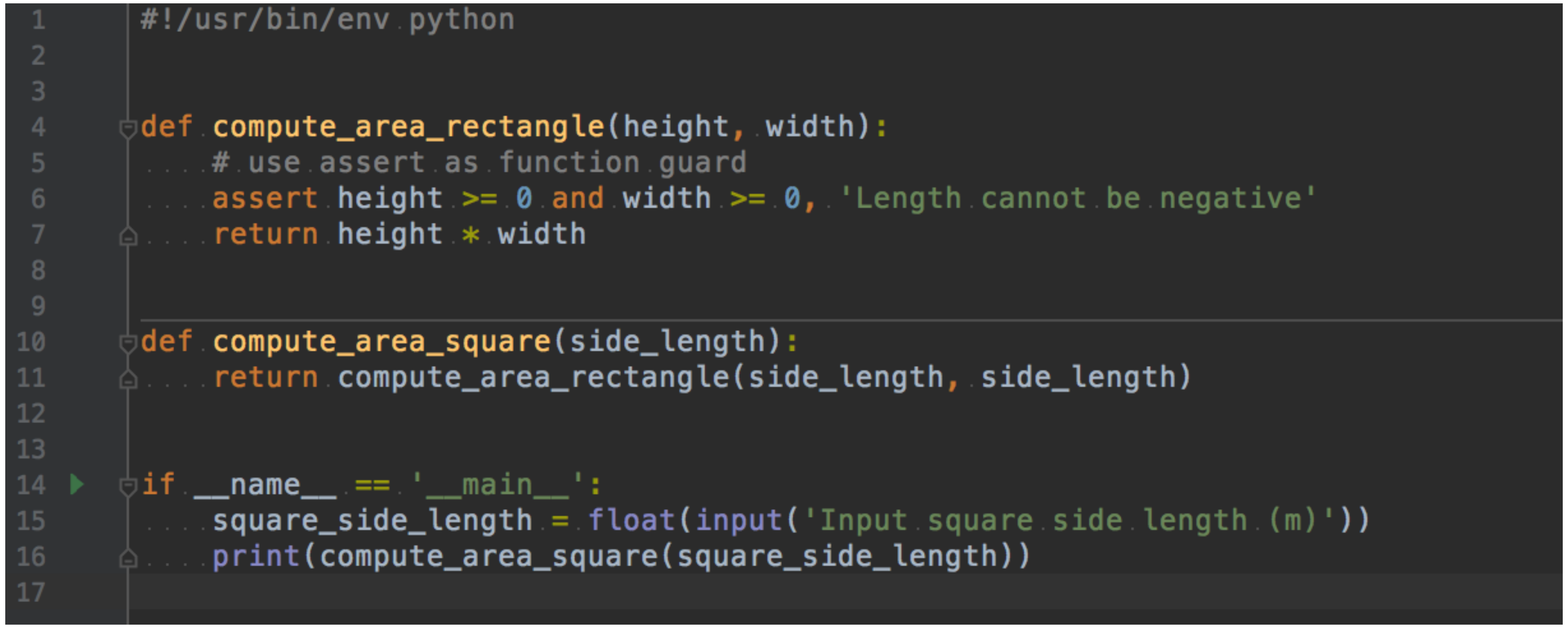
```python
def is_divisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

```python
if is_divisible(x, y):
    ... # Do something ...
else:
    ... # Do something else ...
```

- Functions that return Boolean values
- Give Boolean functions names that sound like yes/no questions, e.g. **is_divisible**
- Condition of the **if** statement is itself a **Boolean expression**

FUNCTIONS CALLING FUNCTIONS

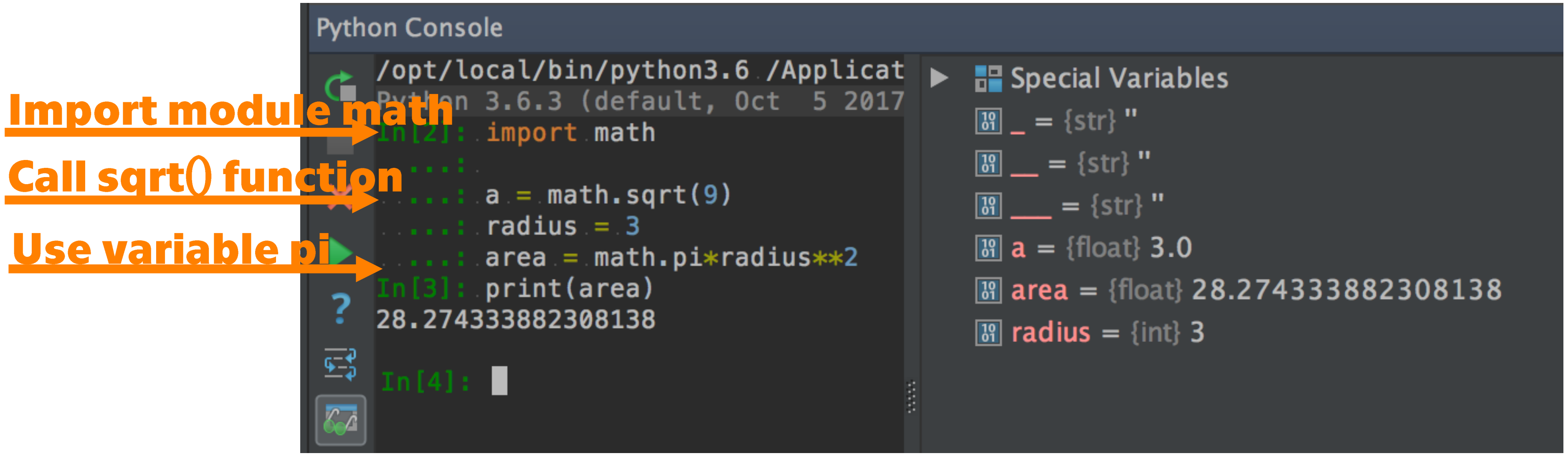


```python
#!/usr/bin/env python


def compute_area_rectangle(height, width):
    # use assert as function guard
    assert height >= 0 and width >= 0, 'Length cannot be negative'
    return height * width


def compute_area_square(side_length):
    return compute_area_rectangle(side_length, side_length)


if __name__ == '__main__':
    square_side_length = float(input('Input square side length (m)'))
    print(compute_area_square(square_side_length))
```

- Functions **hide complex computation** behind a single command and capture abstraction of the problem.
- Functions can **simplify** a program
- Creating a new function can make a **program shorter** by eliminating **repetitive code**

source http://openbookproject.net/thinkcs/python/english3e/functions.html

LIBRARIES, MATH ...



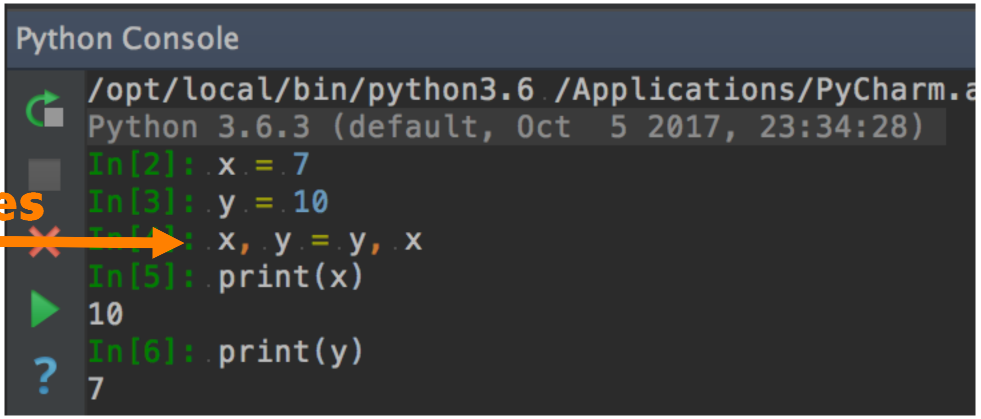
```
/opt/local/bin/python3.6 /Applicat
Python 3.6.3 (default, Oct  5 2017
In[2]: import math
   ...:
   ...: a = math.sqrt(9)
   ...: radius = 3
   ...: area = math.pi*radius**2
In[3]: print(area)
28.274333882308138

In[4]:
```

```
Special Variables
_ = {str} ''
__ = {str} ''
___ = {str} ''
a = {float} 3.0
area = {float} 28.274333882308138
radius = {int} 3
```

- https://docs.python.org/3.4/library/math.html

FLOW OF EXECUTION

Swapping variables

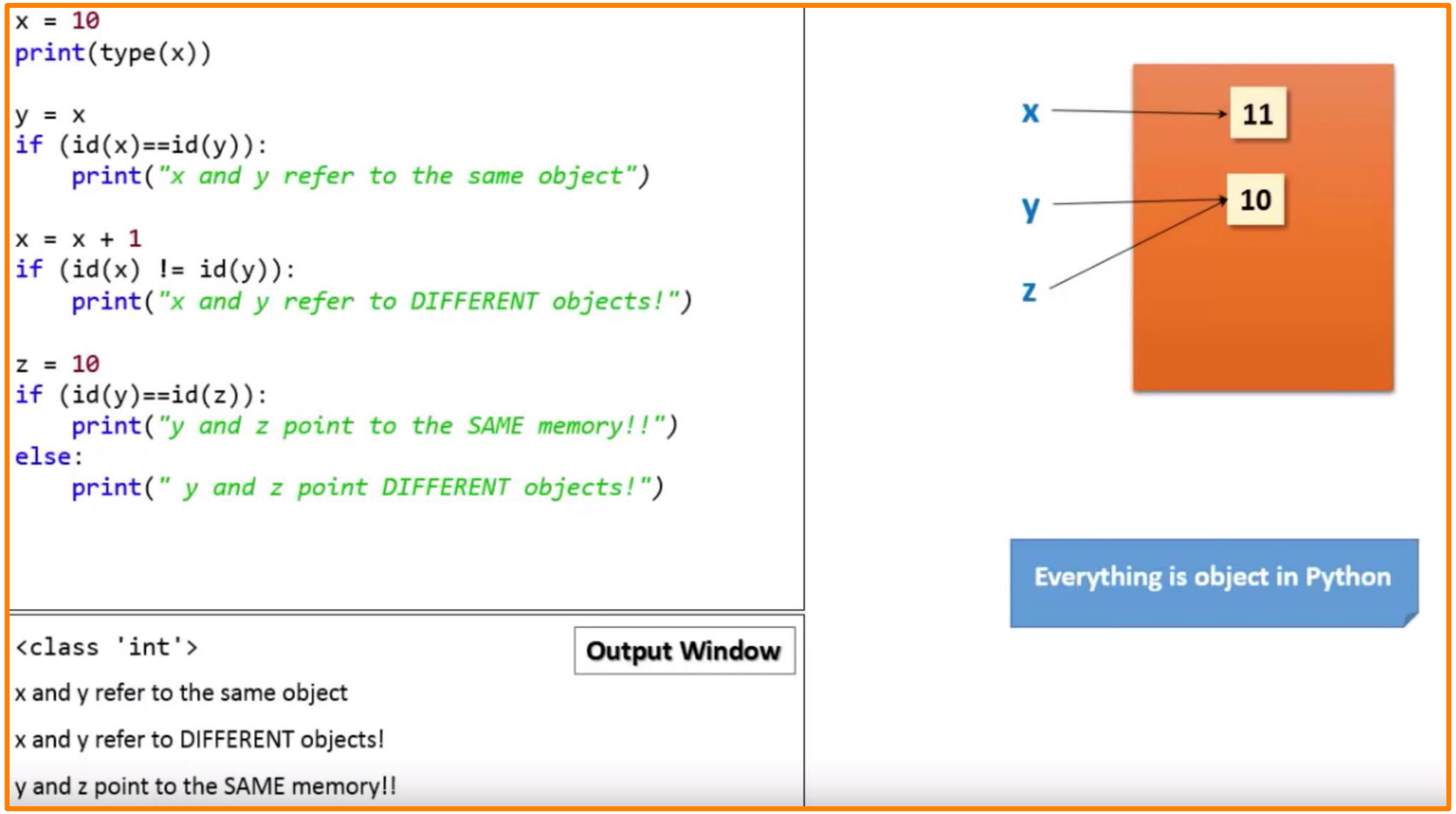


```
/opt/local/bin/python3.6 /Applications/PyCharm.a
Python 3.6.3 (default, Oct  5 2017, 23:34:28)
In[2]: x = 7
In[3]: y = 10
In[4]: x, y = y, x
In[5]: print(x)
10
In[6]: print(y)
7
```

- Flow of execution = **order of statements execution** (begins at the first statement of the program)
- Statements are executed **one at a time**, in order from top to bottom (but **read the flow**, not top to bottom!)
- Python evaluates **expressions from left to right** (during assignment right-hand side is evaluated first)
- Function calls are like a **detour** in the flow of execution
- We can define one function inside another
- Function **definitions do not alter the flow** of execution

source http://docs.python.org/3/reference/expressions.html#evaluation-order

MEMORY



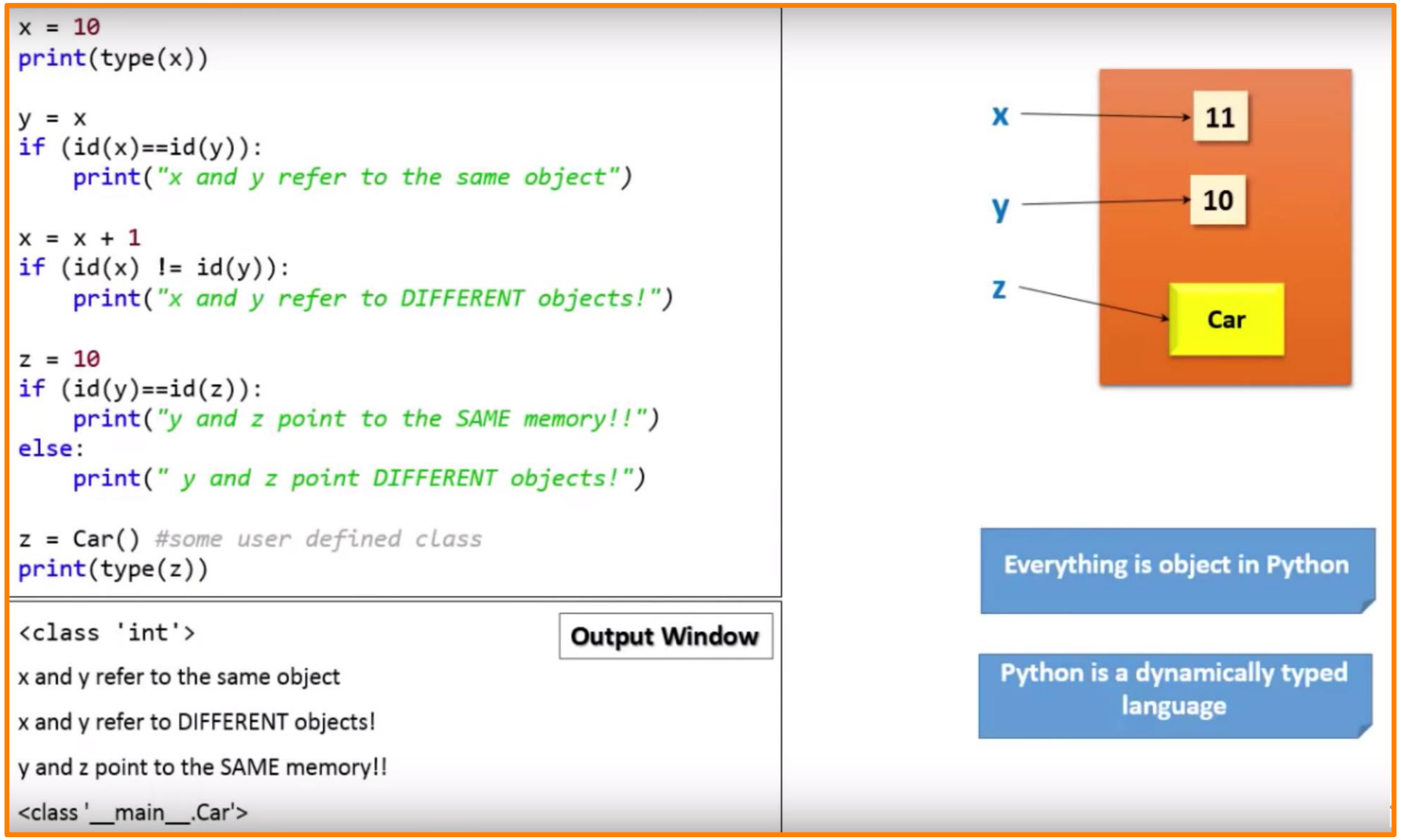
```
x = 10
print(type(x))

y = x
if (id(x)==id(y)):
    print("x and y refer to the same object")

x = x + 1
if (id(x) != id(y)):
    print("x and y refer to DIFFERENT objects!")

z = 10
if (id(y)==id(z)):
    print("y and z point to the SAME memory!!")
else:
    print(" y and z point DIFFERENT objects!")
```

Output Window

```
<class 'int'>
x and y refer to the same object
x and y refer to DIFFERENT objects!
y and z point to the SAME memory!!
```

source https://www.youtube.com/watch?v=arxWaw-E8QQ&t=1s

MEMORY



```python
x = 10
print(type(x))

y = x
if (id(x)==id(y)):
    print("x and y refer to the same object")

x = x + 1
if (id(x) != id(y)):
    print("x and y refer to DIFFERENT objects!")

z = 10
if (id(y)==id(z)):
    print("y and z point to the SAME memory!!")
else:
    print(" y and z point DIFFERENT objects!")

z = Car() #some user defined class
print(type(z))
```

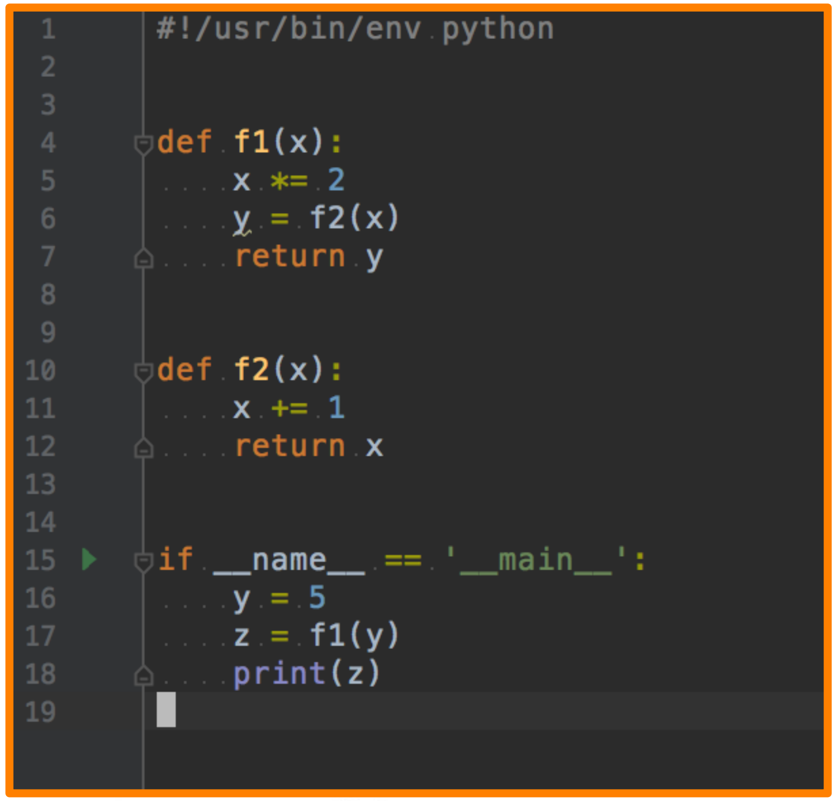
Output Window

```
<class 'int'>
x and y refer to the same object
x and y refer to DIFFERENT objects!
y and z point to the SAME memory!!
<class '__main__.Car'>
```

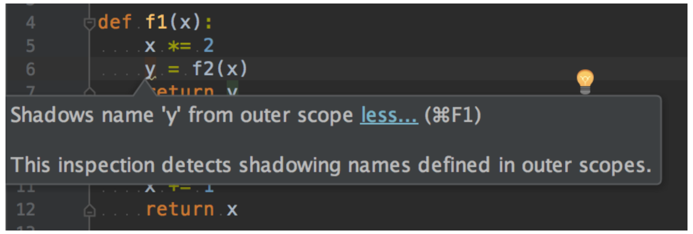
Everything is object in Python

Python is a dynamically typed language

MEMORY



```python
#!/usr/bin/env python


def f1(x):
    x *= 2
    y = f2(x)
    return y


def f2(x):
    x += 1
    return x


if __name__ == '__main__':
    y = 5
    z = f1(y)
    print(z)

```

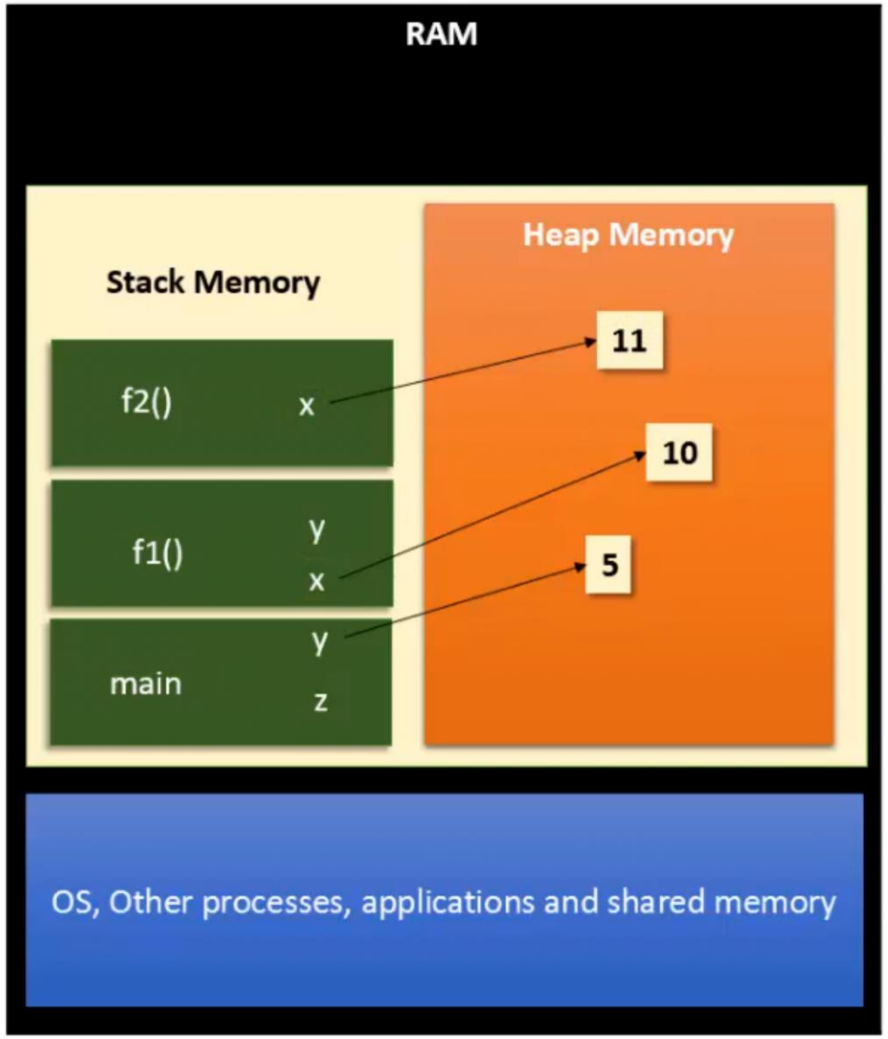


```python
def f1(x):
    x *= 2
    y = f2(x)
    return y
```

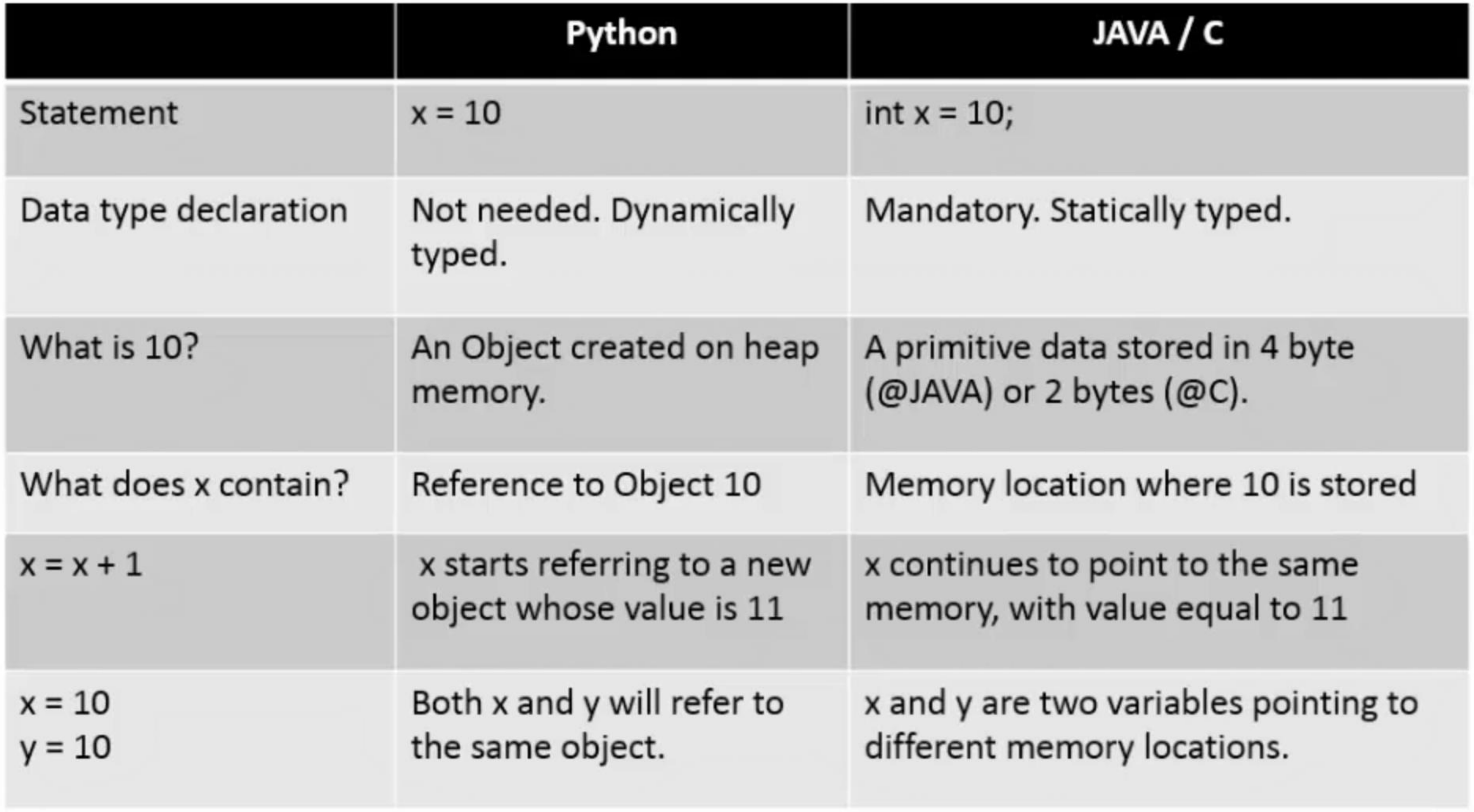
Shadows name 'y' from outer scope less... (⌘F1)

This inspection detects shadowing names defined in outer scopes.

```python
    x += 1
    return x
```



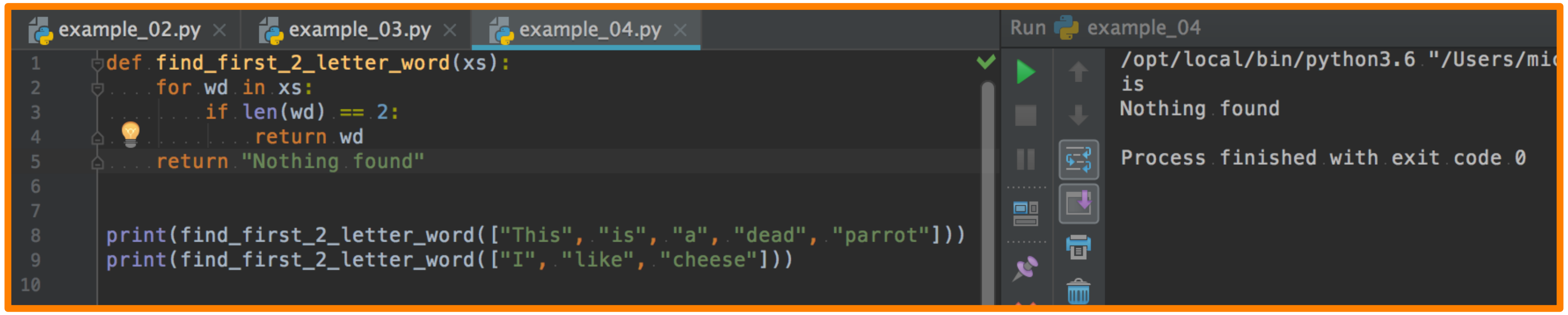
MORE ABOUT PYTHON



	Python	JAVA / C
Statement	x = 10	int x = 10;
Data type declaration	Not needed. Dynamically typed.	Mandatory. Statically typed.
What is 10?	An Object created on heap memory.	A primitive data stored in 4 byte (@JAVA) or 2 bytes (@C).
What does x contain?	Reference to Object 10	Memory location where 10 is stored
x = x + 1	x starts referring to a new object whose value is 11	x continues to point to the same memory, with value equal to 11

x = 10
y = 10
	Both x and y will refer to the same object.	x and y are two variables pointing to different memory locations.

EXAMPLE


example_02.py example_03.py example_04.py

```
def find_first_2_letter_word(xs):
    for wd in xs:
        if len(wd) == 2:
            return wd
    return "Nothing found"


print(find_first_2_letter_word(["This", "is", "a", "dead", "parrot"]))
print(find_first_2_letter_word(["I", "like", "cheese"]))
```

Run example_04

```
/opt/local/bin/python3.6 "/Users/mi
is
Nothing found

Process finished with exit code 0
```

- Return statement in the middle of a **for** loop – control **immediately returns** from the function

- EXAMPLE: *Let us assume that we want a function which looks through a list of words. It should return the first 2-letter word. If there is not one, it should return “Nothing found”*

source http://openbookproject.net/thinkcs/python/english3e/fruitful_functions.html

PROGRAM DEVELOPMENT

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Incremental development** technique – avoid long debugging sessions by adding and testing only a small amount of code at a time.

- EXAMPLE: *We want to find the distance between two points, given by the coordinates (x1, y1) and (x2, y2).* *(**Pythagorean theorem**)*

  What are the inputs (parameters)?
  What is the output (return value)?

PROGRAM DEVELOPMENT

Define interface

```python
def distance(x1, y1, x2, y2):
    return 0.0
```

```
>>> distance(1, 2, 4, 6)
0.0
```

Process parameters

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return 0.0
```

```
>>> distance(1, 2, 4, 6)
0.0
```

Temporary variables

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx*dx + dy*dy
    return 0.0
```

```
>>> distance(1, 2, 4, 6)
0.0
```

Return result

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx*dx + dy*dy
    result = dsquared**0.5
    return result
```

```
>>> distance(1, 2, 4, 6)
5.0
```

PROGRAM DEVELOPMENT

```python
import math

def distance(x1, y1, x2, y2):
    return math.sqrt( (x2-x1)**2 + (y2-y1)**2 )
```

```
>>> distance(1, 2, 4, 6)
5.0
```

- Start with a working **skeleton program** and make small **incremental changes** (analyze errors)
- Use **temporary variables** to refer to intermediate values for easy inspection
- Once the program is working, **explore options** and parameters
- Consolidate multiple statements to make **shorter code**, **refactor for readability**

GLOSSARY

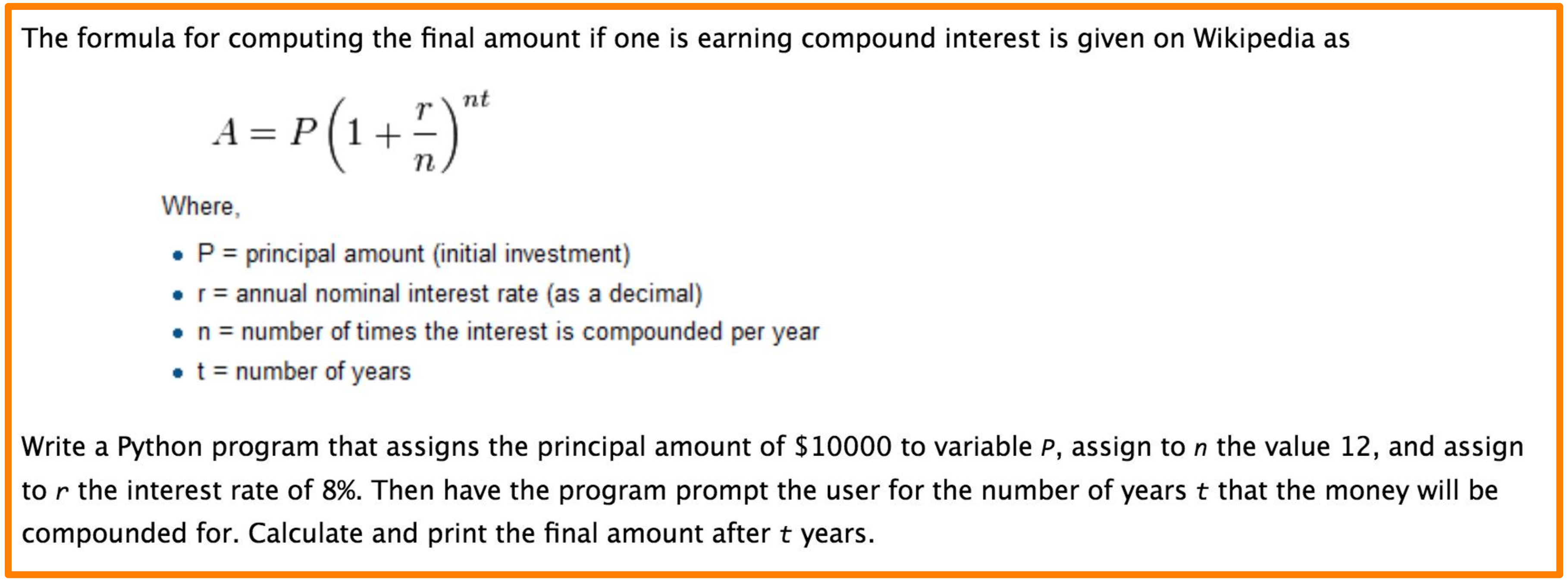
These are the terms you should explore and know:

- Argument
- Header
- Body
- Docstring
- Flow of execution
- Frame
- Function
- Function call
- Function composition
- Function definition
- Fruitful function
- Header line
- Import statement
- Lifetime
- Local variable
- Parameter
- Refactor
- Stack diagram
- Traceback (stack trace)
- void function

Learning with Python 3 - chapter 4.8
http://openbookproject.net/thinkcs/python/english3e/functions.html

source http://openbookproject.net/thinkcs/python/english3e/functions.html

EXAMPLE



The formula for computing the final amount if one is earning compound interest is given on Wikipedia as

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

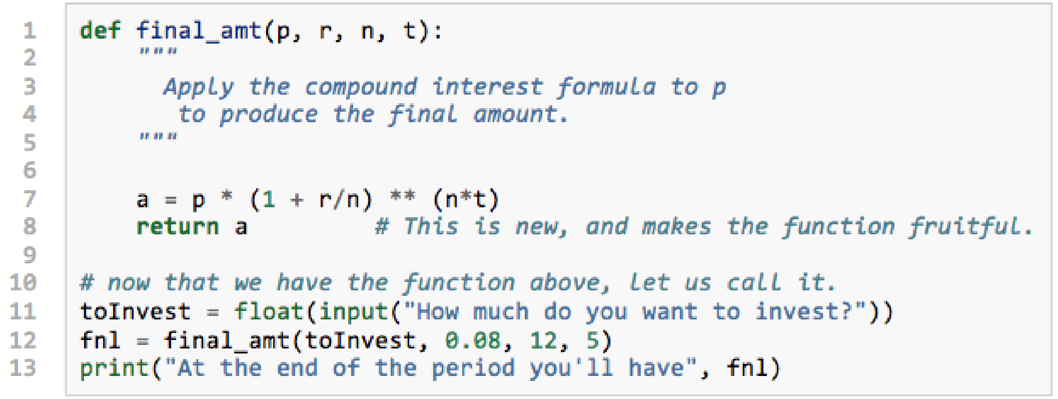
Write a Python program that assigns the principal amount of $10000 to variable P, assign to n the value 12, and assign to r the interest rate of 8%. Then have the program prompt the user for the number of years t that the money will be compounded for. Calculate and print the final amount after t years.

EXAMPLE

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years



```
def final_amt(p, r, n, t):
    """
      Apply the compound interest formula to p
       to produce the final amount.
    """

    a = p * (1 + r/n) ** (n*t)
    return a         # This is new, and makes the function fruitful.

# now that we have the function above, let us call it.
toInvest = float(input("How much do you want to invest?"))
fnl = final_amt(toInvest, 0.08, 12, 5)
print("At the end of the period you'll have", fnl)
```

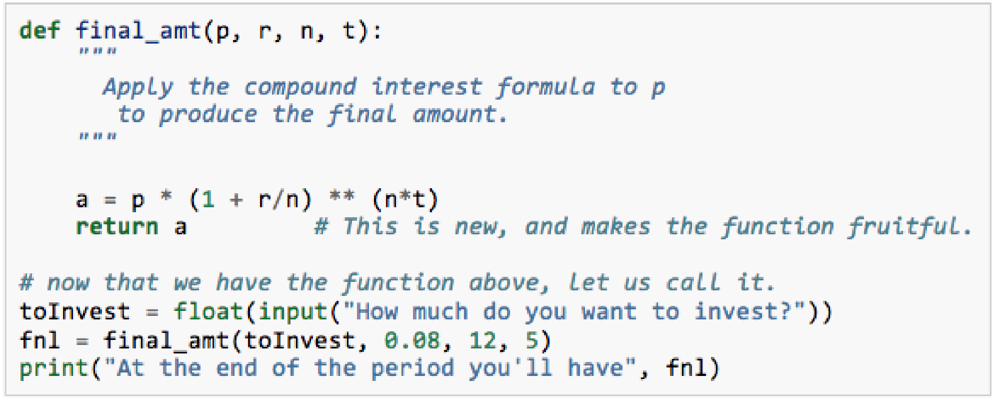
- Will be evaluated and returned to the caller as the “**fruit**”
- Input **prompt** from user (**type conversion** from string to float)
- Arguments for **8%** interest, compounded **12** times per year, for **5** years period
- NOTE: It is as if **p** = **toInvest** is executed when **final_amt** is called (variable name in the caller does not matter, in **final_amt** the name is **p** with **lifetime** until return)

EXAMPLE

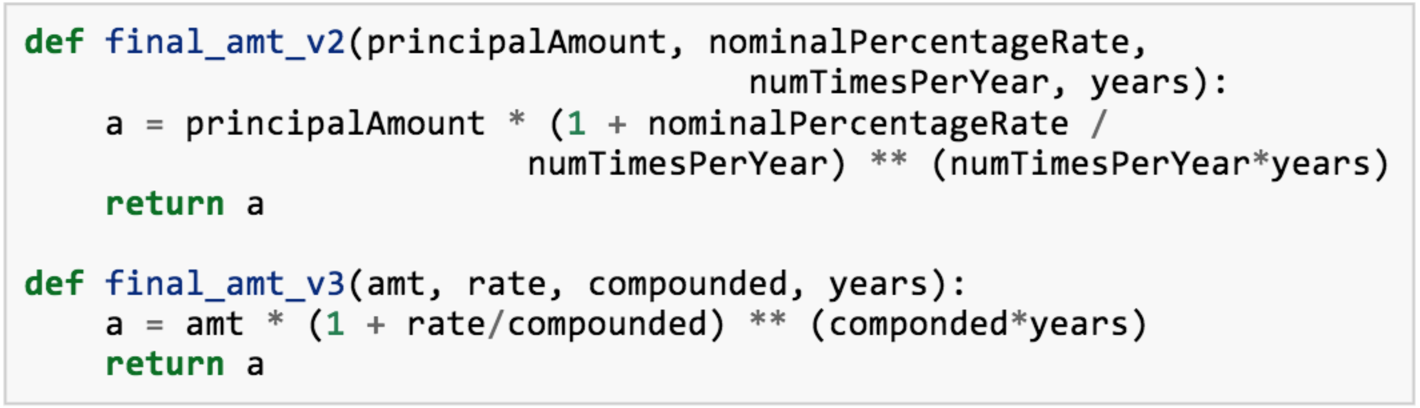
$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years



```python
def final_amt(p, r, n, t):
    """
      Apply the compound interest formula to p
       to produce the final amount.
    """

    a = p * (1 + r/n) ** (n*t)
    return a         # This is new, and makes the function fruitful.

# now that we have the function above, let us call it.
toInvest = float(input("How much do you want to invest?"))
fnl = final_amt(toInvest, 0.08, 12, 5)
print("At the end of the period you'll have", fnl)
```



```python
def final_amt_v2(principalAmount, nominalPercentageRate,
                                    numTimesPerYear, years):
    a = principalAmount * (1 + nominalPercentageRate /
                           numTimesPerYear) ** (numTimesPerYear*years)
    return a

def final_amt_v3(amt, rate, compounded, years):
    a = amt * (1 + rate/compounded) ** (componded*years)
    return a
```

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT
Learning with Python 3 (RLE)
http://openbookproject.net/thinkcs/python/english3e/index.html
available under **GNU Free Documentation License Version 1.3**)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/