Programming

<u>Plan for the rest of the semester:</u>

We will be introducing various new elements of Python and using them to solve increasingly interesting and complex problems.

We saw earlier that computers can (1) input, (2) output, (3) memory (store information), (4) do arithmetic and symbolic processing, and (5) control.

Implication: <u>a program</u> is a sequence of statements (commands and functions) that "speak" to one or more of those capabilities.

The examples we have seen till now used the command shell mode. We enter one statement or expression, hit <enter> and Python evaluates it for us immediately.
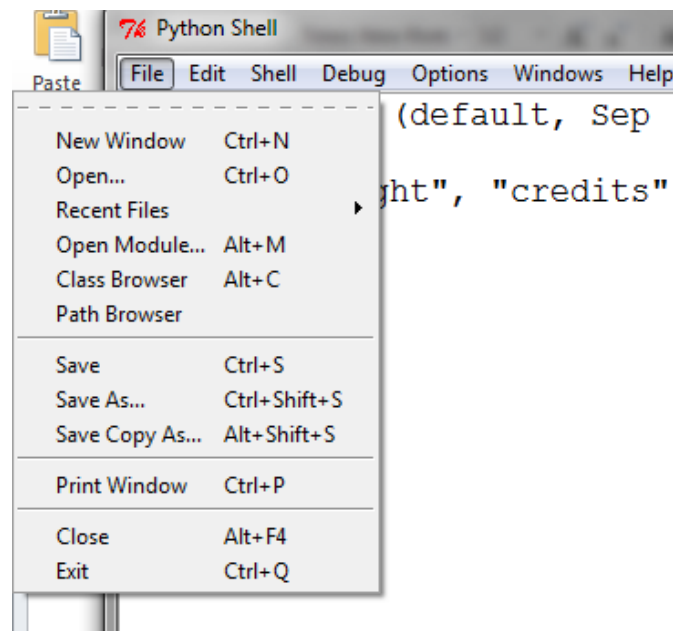
We will start to write programs and get Python to <u>execute</u> (run) them for us.

So … we (1) <u>write</u> then (2) <u>execute</u>.

When we write a program, we

- **create a text file**
- **enter a sequence of statements**
- **when done, we save the file with a .py extension.**

We can write Python programs using any text editor. We will use IDLE (which automatically saves the file with a '.py' extension.
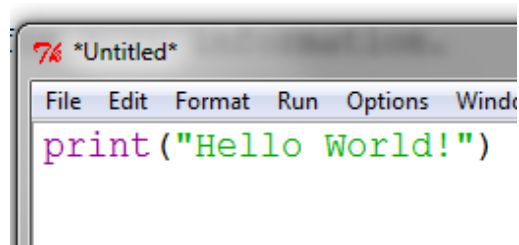
Example: "Hello World" is traditionally the first program everyone writes.
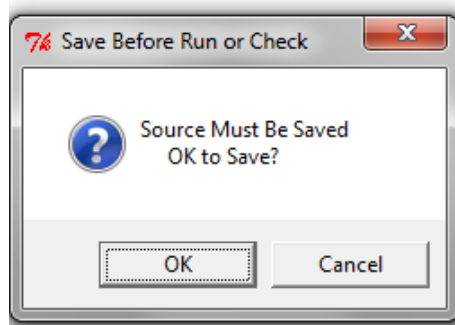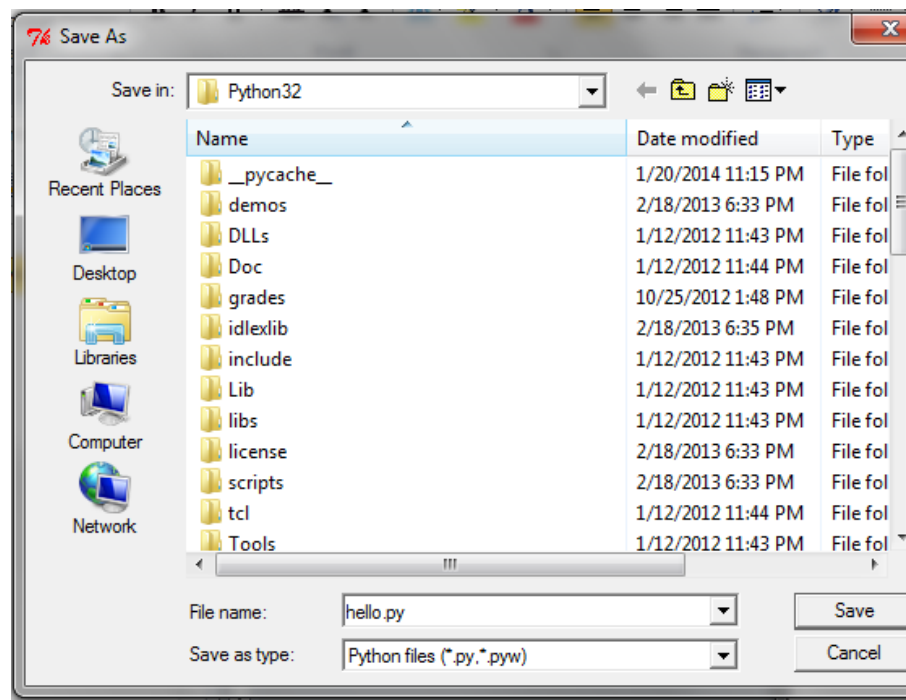
How do we do it?

Answer:
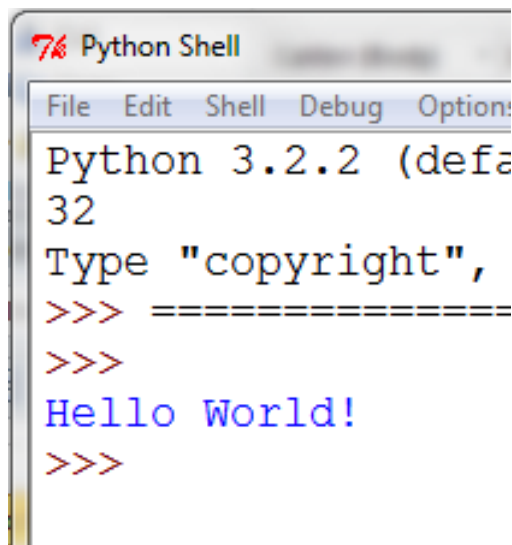 1. Create a new file and type the print command that we saw before.

Pressing F5 will RUN the program.

Type the file name that you want to call the program and Save.

As soon as you Save, the program will run and the result will appear in the interactive shell.

Congratulations! You just wrote your first program!

Using <u>comments</u> in Python

You must admit that this one was pretty simple. Not all programs are that straightforward. As programs get more complex it is often helpful to annotate the various parts of the program so that someone reading it (even you) can figure out what is happening. We do this using "**comments**", which are indicated in Python by the hash symbol "**#**".

They can be of two forms:

They can either be on a line of their own:

```python
# My first program!
print("Hello World!")
```

or at the end of a program statement:

```python
print("Hello World!")# Just wrote my first program!
```

In either case, the Python interpreter ignores the comment. That is the comment doesn't "execute" the comment. It is just there to **help the reader of the program understand** what is going on in the code. This implies that we **shouldn't** comment on things that are **obvious** like:

```python
print("Hello World!")# This is a print statement!
```

New topic: We won't get anywhere interesting without …..

Variables

Variables allow us to **associate a name with a value** so that we can use it later on.

Some examples:

x=5

y=10

z=x+y

These are examples of **assignment statements**. **They have the form:**

variable name = expression

Python evaluates the expression on the right of the = and associates the value obtained with the name on the left.

So in this program:

x=1

y=2

z=x+y

print(x,y,z)

What will this look like in memory?

Trace the program as it runs.

Aside: Using the Python emulator. It will allow us to step through a program and see what happens in the memory as each step is executed.

The emulator may be reached at: http://pythontutor.com/ or from the lecture web page.

LEARN programming by visualizing code execution

Online Python Tutor is a free educational tool created by Philip Guo that helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this tool, a teacher or student can write a Python program in the Web browser and visualize what the computer is doing step-by-step as it executes the program.

As of Dec 2013, over **500,000 people** in over **165 countries** have used Online Python Tutor to understand and debug their programs, often as a supplement to textbooks, lecture notes, and online programming tutorials. Over 6,000 pieces of Python code are executed and visualized *every day*.

Users include self-directed learners, students taking online courses from Coursera, edX, and Udacity, and professors in dozens of universities such as MIT, UC Berkeley, and the University of Washington.

Start using Online Python Tutor now

As a demo, here is a visualization showing a program that recursively finds the sum of a (cons-style) linked list. Click the "Forward" button to see what happens as the computer executes each line of code.

```
1  def listSum(numbers):
2      if not numbers:
3          return 0
4      else:
5          (f, rest) = numbers
6          return f + listSum(rest)
7
8  myList = (1, (2, (3, None)))
9  total = listSum(myList)
```

Frames   Objects

Edit code

< Back   Step 1 of 22   Forward >

→ line that has just executed
➡ next line to execute

Code visualized with Online Python Tutor

The Python emulator will be very helpful in debugging.

What is debugging?

Answer:

When we run it we get the following:

```
1  x=1
2  y=2
3  z=x+y
4  print(x,y,z)
```

Edit code

<< First   < Back   Step 4 of 4   Forward >   Last >>

has just executed
to execute

Frames   Objects

Global frame
x → int 1
y → int 2
z → int 3

Notice how the variables x, y, and z **point to** their respective values.

Python has four other forms of assignment.

1. Extended assignment.

$$x=y=z=5$$

```
1  x=y=z=5
```

Edit code

Program terminated   Forward >   Last >>

Frames   Objects

Global frame
z
y
x → int 5

2. Assignment to tuples (also called "unpacking").

$$x,y,z=1,2,3$$

```
1  x,y,z=1,2,3
```

Edit code

Program terminated   Forward >   Last >>

Frames   Objects

Global frame
x → int 1
y → int 2
z → int 3

3. Augmented assignment statement.

The Python documentation puts it like this:

Augmented assignment is the combination, in a single statement, of a <u>binary operation and an assignment statement</u>:

For example:

Instead of writing

     x=x+1

we can write

     x+=1

where both accomplish the same thing, viz. increment x by one. More generally we have

     x+= expression

which increments x by the value of the expression.

We also have the following augmented assignment operators:

+=

-=

*=

/=

//=

%=

**=

Which do, analogous to +=, what you would expect.

   4. There is also the "walrus" assignment operator (from 3.8 and up)

```
>>> print(a=7)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(a=7)
TypeError: 'a' is an invalid keyword argument for print()
>>> a=7
>>> print(a:=7)
7
>>> |
```

Input

How do we get input from the keyboard to use in our program?

```
name=input (
print ("Hel input([prompt]) -> string
```

Question:

What will the following program print?

print('Hello')
print('My name is <put your name here>') #insert your name

Problem:

Rewrite the above program using only one print statement

Write a program to ask a "user" to type their name. Then have your program greet them.

```
name=input ("Please enter your name: ")
```

Now what??? <u>Write a program</u> to generate the following output. Of course, the entered name should appear instead of "Jerry" (unless you entered "Jerry" … ☺)

```
>>>
please enter your name:Jerry
Hello Jerry !
>>>
```

Answer:

What is going on in the computer to make your program execute correctly?

Problem:

Modify the above program so that the "!" follows the name <u>without an intervening blank</u>.

```
>>>
please enter your name:Jerry
Hello Jerry!
>>>
```

Answer

Problem:

Write a program to prompt the user for two numbers. Your program will then print the sum.

```
>>>
Please enter the first number: 4
Please enter the second number: 5
The sum of 4 and 5 is 9.
>>> |
```

Answer:

Problem:

Modify your answer to the above problem so that we get the following:

```
>>>
Please enter the first number: 5
Please enter the second number: 7
The sum of 5 and 7 is 12.
The difference of 5 and 7 is -2.
The product of 5 and 7 is 35.
The integer division of 5 and 7 is 0.
The float division of 5 and 7 is 0.7142857142857143.
The remainder of dividing 5 by 7 is 5.
>>> |
```

Answer:

Problem:

Write a program to ask the user their age in years. Your program should calculate and print their age in seconds.

Answer:

Problem:

Write a program that asks the user to enter a temperature in Fahrenheit. Your program should calculate and print the temperature in Centigrade.

We can translate between Fahrenheit and Celsius according to the following formulas:

$$°C \times 9/5 + 32 = °F$$

$$(°F - 32) \times 5/9 = °C$$

Problem:

Given a temperature in Centigrade, write the assignment statement to calculate the temperature in Fahrenheit. Now, do it the other way, i.e. go from Fahrenheit to Celsius.

Problem – Rounding a number:

What does it mean to round a number? Say x is a floating-point number (i.e., contains a decimal point). If its fractional part is .5 or greater then we "round" x to the next highest integer. But if the fractional part is less than .5, we "round down" to x. So, rounding 34.56 ➔35 and 34.48➔ 34. **Notice** that when we round a float, we get an int.

Write an assignment statement to convert a floating point number x, to its rounded value y.

Do this in **two** ways: (1) **without** using the built-in "round" function, and (2) once with. Here is how the Python documentation describes the round function.

`round`$(x[, n])$

Return the floating point value *x* rounded to *n* digits after the decimal point. If *n* is omitted, it defaults to zero.

Note: The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See *Floating Point Arithmetic: Issues and Limitations* for more information.

Answer:

Note that the first method works correctly for positive numbers but has different behavior for negative numbers compared to the round function. <u>The round function rounds to the nearest even number when the number is exactly halfway between two integers.</u>

For example:

Consider the floating-point number **-2.5**. We want to round this number to the nearest integer.

1.**Using int(x + 0.5)**:
    1. When we apply this method to **-2.5**, we add **0.5** to the number, resulting in **-2.0**. Then, applying **int()** to **-2.0** gives **-2**.
    2. So, **-2.5** becomes **-2**.
2.**Using round(x)**:
    1. The **round** function in Python rounds half to even (also known as "**bankers' rounding**"). This means that when a number is exactly halfway between two integers, it rounds to the nearest even integer.
    2. In this case, **-2.5** is exactly halfway between **-2** and **-3**. According to Python's rounding rule, it rounds to the nearest even number, which is **-2**.
    So, **round(-2.5)** also gives **-2**.

In this particular example of **-2.5**, both methods yield the same result, **-2**. However, the difference becomes apparent with numbers like **-1.5**.

1.**Using int(x + 0.5)**:
    * For **-1.5**, adding **0.5** results in **-1.0**. Using **int()** on **-1.0** yields **-1**.
2.**Using round(x)**:
    1. **round(-1.5)** will round to the nearest even number, which is **-2** in this case.

Thus, for **-1.5**, the first method gives **-1**, while the **round** function gives **-2**. This demonstrates the different behaviors of the two methods when dealing with negative numbers that are exactly halfway between two integers. The **int(x + 0.5)** method always rounds away from zero, while the **round** function employs bankers' rounding.

What about for positive numbers?

Well, let x=2.5.

3.**Using int(x + 0.5)**:
    * For **2.5**, adding **0.5** results in **3.0**. Using **int()** on **3.0** yields **3**.
4.**Using round(x)**:
    1. **round(2.5)** will round to the nearest even number, which is **2** in this case.

Problem:

Write a program that asks the user to enter an integer. Your program should "echo" the input and print True if the number is even and False if the number is odd.

Problem:

Write a program that asks the user to enter an integer. Your program should "echo" the input and print EVEN if the number is even and ODD if the number is odd.

Problem:

Write a program that asks the user to enter two integers. Your program should:

- "Store" the values entered in variables a and b.
- "Print a and b.
- "Swap" the values in a and b, so that what was in a is now in b and vice versa.
- Print a and b. At this point the values should be the reverse of what you printed above.

Answer:

Will this one work?

How about this?

Note: Python allows a more direct way to do this using "unpacking." Here is how:

How come this works?

41

Problem:

Look at the following:

```
>>>
Please enter the number of hours that you worked: 5
Please enter your hourly rate: 10
You earned 50.0 for 5.0 work at 10.0 dollars/hour.
>>> ============================== RESTART ==============
>>>
Please enter the number of hours that you worked: 56
Please enter your hourly rate: 45
You earned $ 2520.0 for 56.0 work at 45.0 dollars/hour.
>>> ============================== RESTART ==============
>>>
Please enter the number of hours that you worked: 45
Please enter your hourly rate: 67
You earned $ 3015.0 for 45.0 work at $ 67.0 dollars/hour.
>>> ============================== RESTART ==============
>>>
Please enter the number of hours that you worked: 67
Please enter your hourly rate: 98
You earned $6566.0 for 67.0 work at $98.0 dollars/hour.
>>>
```

Write a program that will calculate a worker's pay given the number of hours worked and the rate per hour. Both values could be floats, for example if someone worked 35.5 hours at a rate of $35.75 an hour. Work through the example above.

How can we format the output?

Let's see three ways.

1. The Format function

This function takes **two arguments**:

- The value that you want printed
- The specification of the format to use – for example "field width", number of digits after the decimal point, etc.

The syntax is:

format(value to be printed, format specification)

Here is the Python documentation description:

format(*value*[, *format_spec*])

> Convert a *value* to a "formatted" representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument, however there is a standard formatting syntax that is used by most built-in types: *Format Specification Mini-Language*.

The general form of a standard format specifier is:

```
format_spec ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill        ::=  <a character other than '}'>
align       ::=  "<" | ">" | "=" | "^"
sign        ::=  "+" | "-" | " "
width       ::=  integer
precision   ::=  integer
type        ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

What in the world does this mean???

We will look at some of the most common ones.

Examples

Formatting floating point numbers:

```
>>>
>>> print(format(123456.789,'.2f'))
123456.79
>>> print(format(123456.789,'20.2f'))
            123456.79
>>> print(format(123456.789,'20,.2f'))
          123,456.79
>>> print(format(123456.789,'e'))
1.234568e+05
>>> print(format(123456.789,'20.2e'))
            1.23e+05
>>>
```

43

What does the "+" specify?

```
"""
>>> print(format(123456.789,'+20.2f'))
          +123456.79
>>> print(format(123456.789,'-20.2f'))
           123456.79
```

Formatting integers:

```
>>> print(format(123456,'20d'))
              123456
>>> print(format(123456,'<20d'))
123456
>>> print(format(123456,'>20d'))
              123456
>>> print(format(123456,'^20d'))
       123456
>>>
```

Formatting strings:

```
>>>
>>> print(format('abcd','20s'))
abcd
>>> print(format('abcd','<20s'))
abcd
>>> print(format('abcd','>20s'))
                abcd
>>> print(format('abcd','^20s'))
        abcd
>>>
```

The format is used to control the formatting of individual items.

    2. For more complex string formatting, you can use <u>f-strings</u>:

x = 123.456

f'Value is {x:0.2f}'   # 'Value is 123.46'
f'Value is {x:10.4f}'   # 'Value is   123.4560'
f'Value is {2*x:*<10.2f}' # 'Value is 246.91****'

How does it work?

Within an f-string, **text of the form {expr:spec}** is replaced by the value of **format(expr, spec)**. expr can be an arbitrary expression as long as it doesn't include {, }, or \ characters.

Parts of the format specifier itself can optionally be supplied by other expressions.

For example:

y = 3.1415926
width = 8
precision=3

r = f'{y:{width}.{precision}f}' # r = ' 3.142'

If you end expr by =, then the literal text of expr is also included in the result.

For example:

x = 123.456

f '{x=:0.2f}'   # 'x=123.46'
f'{2*x=:0.2f}'   # '2*x=246.91'

If you append !r to a value, formatting is applied to the output of **repr().**

If you use !s, formatting is applied to the output of **str().**

For example:

f'{x!r:spec}'   # Calls (repr(x).__format__('spec'))
f'{x!s:spec}'   # Calls (str(x).__format__('spec'))

For example:

```
f'{x!r:spec}'      # Calls (repr(x).__format__('spec'))
f'{x!s:spec}'      # Calls (str(x).__format__('spec'))
```

So … what's the difference between str and repr?

repr gives us more information that is helpful in debugging.

Although str() and repr() both create strings, their output is often different. str() produces the output that you get when you use the print() function, whereas repr() creates a string that you type into a program to exactly represent the value of an object. For example:

```
>>> s = 'hello\nworld'
>>> print(str(s))
hello
world
>>> print(repr(s))
'hello\nworld'
>>>
```

When debugging, use repr(s) to produce output because it shows you more information about a value and its type.

3. As an alternative to f-strings, we can use the .format() method of strings:

x = 123.456

```
'Value is {:0.2f}' .format(x)          # 'Value is 123.46'
'Value is {0:10.2f}' .format(x)        # 'Value is   123.4560'
'Value is {val:<*10.2f}' .format(val=x) # 'Value is 123.46****'
```

With a string formatted by .format(), text of the form {arg:spec} is replaced by the value of format(arg, spec).

In this case, arg refers to one of the arguments given to the format() method. If omitted entirely, the arguments are taken in order.

For example:

```
name = 'IBM'
shares = 50
price = 490.1
```

r = '{:>10s} {:10d} {:10.2f}'.format(name, shares, price)
this will produce r = '   IBM   50   490.10'

see "What is the difference between __str__ and __repr__ in Python?"
for an excellent extended explanation.

arg can also refer to a specific argument number or name.

For example:

```
tag = 'p'
text = 'hello world'

r = '<{0}>{1}</{0}>'.format(tag, text)  # r = '<p>hello world</p>'
r = '<{tag}>{text}</{tag}>'.format(tag='p', text='hello world')
```

Unlike f-strings, the arg value of a specifier cannot be an arbitrary expression, so it's not quite as expressive.

However, the format() method can perform limited attribute lookup, indexing, and nested substitutions.

For example:

```
y = 3.1415926
width = 8
precision=3

r = 'Value is {0:{1}.{2}f}'.format(y, width, precision)

d = {
   'name': 'IBM',
   'shares': 50,
   'price': 490.1
}
r = '{0[shares]:d} shares of {0[name]} at {0[price]:0.2f}'.format(d)
# r = '50 shares of IBM at 490.10'
```

Problem:

Let's modify the payment problem above so that we now get the following:

```
///
Please enter the number of hours that you worked: 45.6
Please enter your hourly rate: 56.7
You earned $2585.52 for 45.6 hours of work at $56.70/hour.
>>>
```

Answer:

Problem

Write a program to provide **change in coins.**

<u>Input</u>

Have your program ask the user to input some amount of money.

<u>Output</u>

Your program should output the minimum number of coins required to render the amount entered into coins. The coins are half-dollars, quarters, dimes, nickels and pennies.

Your program should also list how many of each coin needs to be provided. Each denomination on its own line, starting with half-dollars.

Problem: **Salary Calculation (no if statements)**

Write a program that **prompts your user** for

  1. the number of hours worked and

  2. the hourly rate,

and calculates the amount payable given the entered values.

Output

Your program should output **three lines**:

  1. "Straight pay" = the amount owed to worker for work less than or equal to 40 hours. 2. "Overtime pay"

= the amount owed to the worker for work done after 40 hours 2. "Total pay" =the sum of 1 and 2 above.

Assumptions

The normal workweek is 40 hours.

If the worker worked **forty hours or less,** they are compensated at the rate times the number of hours worked.

 If the worker worked more than forty hours, any hours worked beyond 40 is compensated at "time an-a-half." "Time-an-a-half" means 1.5 times the 40-hour rate. For example, if someone works for 50 hours at $10/hr they earn 40*10+10*15 =$550.

New topic

Recall:

1. Input
2. Output
3. Memory
4. Arithmetic – symbolic manipulation and evaluation
5. Control

We have seen examples of 1 – 4.

What about …

1. Input
2. Output
3. Memory
4. Arithmetic – symbolic manipulation and evaluation

5. Control

We have seen a simple example of control. Python simply executes one statement after another from the first statement of the program to the last. However, sometimes we want to execute statements **conditionally**.

To do this we use the

if statement

There are three forms. Here is the first..

```
age=int(input("What is your age: "))
if age > 99:
    print("Very funny.")
minutes = age * 365 * 24 * 60
print("You are ",minutes, " minutes old.")
```

When the above program is run (twice) we get:

```
>>>
What is your age: 789
Very funny.
You are  414698400  minutes old.
>>> =============================
>>>
>>>
What is your age: 45
You are  23652000  minutes old.
>>>
```

The **syntax** (form) of the if statement is:

if \<Boolean expression> :
one or more statements (called a code block)

According to the Python documentation, a code block:

A *block* is a piece of Python program text that is executed as a unit.

The **semantics** (meaning or interpretation) is:

1. Evaluate the Boolean expression.
2. If its value is True then execute the indented code block, and then continue on with the statement after the if.
3. If its value is False then simply skip the indented code block and continue with the statement after the if.

Problem:

Write a program that inputs a number and determine if the number is even or odd.

```
>>>
Please enter an integer: 68
68 is even.
>>> ============================
>>>
Please enter an integer: 67
67 is odd.
>>> |
```

Do this in **two** ways. (1) Using two if statements. (2) Using only one if statement and no "fancy" Boolean expressions.

Answer:

Problem:

Write a program that asks for a number, 1, 2, or 3.

- If a 1 is entered print the color is red!
- If a 2 is entered print the color is green!
- If a 3 is entered print the color is blue!
- If the number entered is none of those, print ERROR!

Answer

Problem:

- If n is in the range 1-10, print the color is red!
- If n is in the range 1-10, print the color is red!
- If n is in the range 11-20, print the color is blue!
- If n is in the range 21-30, print the color is green!
- If the number entered is none of those, print ERROR!

Answer:

if statement – second form

Example.

```
n=int(input('Please enter an integer: '))
if n%2==0:
    print(n,'is even.')
else:
    print(n,'is odd.')
```

The **syntax** (form) of this if statement is:

> **if <Boolean expression> :**
> **one or more statements (called a code block)**
> **else:**
> **one or more statements (called a code block)**

The **semantics** (meaning or interpretation) is:

1. Evaluate the Boolean expression.
2. If its value is True then execute the indented code block, and then continue on with the statement after the if.
3. If its value is False then execute the indented code block statement after the else.

Note: The indentation is important in Python. If the blocks in the "if statement" were not indented to the right of the "if", Python would call you out on a syntax error.

How much to indent? Python doesn't care, but the accepted style in the "community" is four spaces.

Problem:

Write a program that inputs **two** different integers and prints out the larger of the two.

Answer:

Problem:

Write a program that inputs **three** different integers and prints out the largest of the three.

Answer:

Problem:

Write a program that inputs **<u>five</u>** different integers and prints out the largest of the five.

Answer:

Problem:

Re-write the following if statement so that it is syntactically correct; then figure out what will be printed. Test your answer by running the reformatted code.

```
x = 9
y = 8
z = 7
```

```
if x > 9: if y > 8: print ("x > 9 and y > 8")  else: if z >= 7: print("x <= 9 and z >= 7") else: print("x <= 9 and z < 7")
```

In many programming languages the general rule regarding" nested" if statements (an if in the block of another if) is that the "else" goes with the closest "if" that has no "else."

Answer:

Like C++, Python also has a Ternary Operator (conditional operator)

It's like an if else.

```
int main() {
  int number = -4;
  string result;

  // Using ternary operator

  result = (number > 0) ? "Positive Number!" : "Negative Number!";

  cout << result << endl;
  return 0;
}
```

In python it looks like this:

```
<variable> = <true_value> if <condition> else <false_value>
```

Here's an example:

```
a = int(input())
b = int(input())
# check which is smaller

x = a if a<b else b

print("Smaller Number is: ",x)
```

The following strange syntax also works with tuples and lists:

```
 (false_value,true_value)[condition]
```

Can you figure out what's happening?

```
# Take input of two numbers
a = int(input())
b = int(input())
# check which is smaller using tuples
# (if false,if true)[condition]
print("Smaller Number is: ",(b,a)[a<b])
print("Smaller Number is: ",[b,a][a<b])
```

if statement – third form

```python
n=int(input('Please enter an integer between 1 and 5: '))
if n==1:
    print('Red!')
elif n==2:
    print('Green!')
elif n==3:
    print('Blue!')
elif n==4:
    print('Indigo!')
elif n==5:
    print('Violet!')
else:
    print("ERROR!")
```

The **syntax** (form) of this if statement is:

if <**Boolean expression**> **:**
   one or more statements (called a code block)

elif <**Boolean expression**> **:**
   one or more statements (called a code block)

elif <**Boolean expression**> **:**
   one or more statements (called a code block)

.
.
.

else:
   one or more statements (called a code block)

Note: The final "else" is optional.

Question: How many "elif"'s do you need?

Answer: As many as you need for the problem that you are solving.

The **semantics** (meaning or interpretation) is:

1. Starting at the first if, find the first Boolean expression that evaluated to True.
2. If one is found, execute the block of statement associated with that if, then exit the whole "if construct" and continue the program with the statement following it.
3. If no Boolean expression evaluates to True
   a. If there is an "else clause", evaluate it and execute the block of statement associated with it. Then continue the program with the statement following the else cause.
   b. If there is no "else clause", then exit the whole "if construct" and continue the program with the statement following it.

Problem:

Write a program using "if – elif" that converts a numerical score to a letter grade according to the following scheme:

- If the score is 90 or above, the grade is A.
- If the score is between 80 and 90 the grade is B.
- If the score is between 70 and 80 the grade is C.
- If the score is between 60 and 70 the grade is D.
- If the score is less than 60, the grade is F.

In the above**, between x and y** means starting with x and less than y.

The program prints the score and the associated letter grade.

If the score entered is not in any of the above ranges, the program prints the score and the message "Score out of Range!".

Answer:

Problem:

We will say that a number is a **palindrome** if it reads the same forwards and backwards. So:

- 1221 is a palindrome
- 6556 is a palindrome
- 1234 is not a palindrome

Write a program that inputs a four digit integer and determines if the number is a palindrome or not. Your program should reject (politely) any input not in the appropriate range.

Answer:

Problem:

Write a program that inputs a <u>three digit integer</u> n (**<u>no zero in the units position</u>**) and returns the digits of n in reverse order. Your program should reject (politely) any input not in the appropriate range.

For example if you enter the integer 123, the program should output the **<u>digits</u>** 321 one next to the other.

Answer:

Problem:

Write a program that inputs a three digit integer n (**<u>no zero in the units position</u>**) and returns the integer made up of the digits of n in reverse order. Your program should reject (politely) any input not in the appropriate range.

For example if you enter the integer 123, the program should output the **<u>integer</u>** 321 (not just

Answer: