built in Exception classes

Here's a list of some commonly used built-in exception classes in Python along with examples:

1. **`Exception`**: The base class for all built-in exceptions.

   ```python
   try:
       raise Exception("This is a generic exception.")
   except Exception as e:
       print(f"Caught an exception: {e}")
   ```

2. **`ValueError`**: Raised when a function receives an argument of the correct type but with an invalid value.

   ```python
   try:
       int("abc")
   except ValueError as ve:
       print(f"ValueError: {ve}")
   ```

3. **`TypeError`**: Raised when an operation is performed on an object of an inappropriate type.

   ```python
   try:
       result = 10 + "5"
   except TypeError as te:
       print(f"TypeError: {te}")
   ```

4. **`ZeroDivisionError`**: Raised when the second argument of a division or modulo operation is zero.

   ```python
   try:
       result = 10 / 0
   except ZeroDivisionError as e:
       print(f"ZeroDivisionError: {e}")
   ```

5. **`IndexError`**: Raised when a sequence subscript is out of range.

   ```python
   try:
       my_list = [1, 2, 3]
       value = my_list[10]
```

```
except IndexError as e:
    print(f"IndexError: {e}")
```

6. **`FileNotFoundError`**: Raised when a file or directory is requested but cannot be found.

```python
try:
    with open('nonexistent.txt', 'r') as file:
        content = file.read()
except FileNotFoundError as e:
    print(f"FileNotFoundError: {e}")
```

7. **`KeyError`**: Raised when a dictionary key is not found.

```python
try:
    my_dict = {'key1': 'value1'}
    value = my_dict['nonexistent_key']
except KeyError as e:
    print(f"KeyError: {e}")
```

8. **`AttributeError`**: Raised when an attribute reference or assignment fails.

```python
try:
    length = (10).length
except AttributeError as e:
    print(f"AttributeError: {e}")
```

9. **`AssertionError`**: Raised when an `assert` statement fails.

```python
x = 5
try:
    assert x > 10, "x should be greater than 10"
except AssertionError as e:
    print(f"AssertionError: {e}")
```

10. **`SyntaxError`**: Raised when the parser encounters a syntax error.

```python
try:
    eval("print('Hello, World!'")
except SyntaxError as se:
```

```
    print(f"SyntaxError: {se}")
```

11. **`RuntimeError`**: Raised when an error is detected that doesn't fall into any specific category.

```python
try:
    raise RuntimeError("This is a runtime error.")
except RuntimeError as re:
    print(f"RuntimeError: {re}")
```

12. **`EOFError`**: Raised when the `input()` function hits an end-of-file condition without reading any data.

```python
try:
    input_data = input("Enter something: ")
    raise EOFError("End of file reached unexpectedly.")
except EOFError as eofe:
    print(f"EOFError: {eofe}")
```

13. **`StopIteration`**: Raised by the `next()` function to indicate that there is no further item to be returned by the iterator.

```python
my_iterator = iter([1, 2, 3])
try:
    while True:
        item = next(my_iterator)
except StopIteration:
    print("StopIteration: No more items in the iterator.")
```

14. **`NotImplementedError`**: Raised when an abstract method that needs to be implemented in a subclass is not actually implemented.

```python
class MyBaseClass:
    def my_abstract_method(self):
        raise NotImplementedError("This method must be implemented in the subclass.")

class MyDerivedClass(MyBaseClass):
    pass

obj = MyDerivedClass()
obj.my_abstract_method()
```

15. **`MemoryError`**: Raised when an operation runs out of memory.

```python
try:
    big_list = [0] * 10**9  # Creating a very large list
except MemoryError as me:
    print(f"MemoryError: {me}")
```

16. **`OverflowError`**: Raised when the result of an arithmetic operation is too large to be expressed within the available numeric range.

```python
try:
    result = 10**500
except OverflowError as oe:
    print(f"OverflowError: {oe}")
```

17. **`RecursionError`**: Raised when the maximum recursion depth is exceeded.

```python
def infinite_recursion():
    return infinite_recursion()

try:
    infinite_recursion()
except RecursionError as re:
    print(f"RecursionError: {re}")
```

18. **`EnvironmentError`** (deprecated): Previously used for file I/O errors, but it's now an alias for `OSError`.

19. **`OSError`**: Base class for I/O errors.

```python
try:
    with open('/nonexistent/file.txt', 'r') as file:
        content = file.read()
except OSError as oe:
    print(f"OSError: {oe}")
```

20. **`BlockingIOError`**: Raised when an operation would block on an object (e.g., socket or file) set for non-blocking operation.

```python
```

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    server_socket.bind(('localhost', 8080))
except BlockingIOError as bioe:
    print(f"BlockingIOError: {bioe}")
```

21. **`ConnectionError`**: Base class for connection-related errors.

```python
try:
    raise ConnectionError("This is a connection error.")
except ConnectionError as ce:
    print(f"ConnectionError: {ce}")
```

22. **`ChildProcessError`**: Raised when an operation on a child process fails.

```python
import subprocess

try:
    subprocess.run(['nonexistent_command'])
except subprocess.CalledProcessError as cpe:
    print(f"CalledProcessError: {cpe}")
```

23. **`FileExistsError`**: Raised when trying to create a file or directory that already exists.

```python
try:
    with open('existing_file.txt', 'x'):
        pass
except FileExistsError as fee:
    print(f"FileExistsError: {fee}")
```

24. **`IsADirectoryError`**: Raised when a file operation (such as opening or removing) is requested on a directory.

```python
try:
    with open('existing_directory', 'r'):
        pass
except IsADirectoryError as iade:
    print(f"IsADirectoryError: {iade}")
```

25. **`NotADirectoryError`**: Raised when a directory operation (such as listing) is requested on something that is not a directory.

```python
try:
    files = os.listdir('nonexistent_directory')
except NotADirectoryError as nade:
    print(f"NotADirectoryError: {nade}")
```

26. **`PermissionError`**: Raised when trying to perform an operation that requires specific access permissions.

```python
try:
    with open('/protected/file.txt', 'w'):
        pass
except PermissionError as pe:
    print(f"PermissionError: {pe}")
```

27. **`ProcessLookupError`**: Raised when a given process cannot be found.

```python
try:
    os.kill(9999, signal.SIGTERM)
except ProcessLookupError as ple:
    print(f"ProcessLookupError: {ple}")
```

28. **`TimeoutError`**: Raised when an operation times out.

```python
import time

try:
    time.sleep(5)
    raise TimeoutError("Operation took too long.")
except TimeoutError as te:
    print(f"TimeoutError: {te}")
```

29. **`UnicodeError`**: Base class for Unicode-related errors.

```python
try:
    s = b'\x80'.decode('utf-8')
```

```
except UnicodeError as ue:
    print(f"UnicodeError: {ue}")
```

30. **`UnicodeDecodeError`**: Raised when decoding a Unicode object fails.

```python
try:
    s = b'\x80'.decode('utf-8')
except UnicodeDecodeError as ude:
    print(f"UnicodeDecodeError: {ude}")
```

31. **`UnicodeEncodeError`**: Raised when encoding a Unicode object fails.

```python
try:
    b = '©'.encode('ascii')
except UnicodeEncodeError as uee:
    print(f"UnicodeEncodeError: {uee}")
```

32. **`UnicodeTranslateError`**: Raised when a Unicode translation operation fails.

```python
try:
    'ä'.encode('ascii', 'strict')
except UnicodeTranslateError as ute:
    print(f"UnicodeTranslateError: {ute}")
```

33. **`ModuleNotFoundError`**: Raised when trying to import a module that cannot be found.

```python
try:
    import nonexistent_module
except ModuleNotFoundError as mne:
    print(f"ModuleNotFoundError: {mne}")
```

34. **`ImportError`**: Raised when an import statement fails to find a name that is defined in the module.

```python
try:
    from nonexistent_module import some_function
except ImportError as ie:
    print(f"ImportError: {ie}")
```

35. **`ResourceWarning`**: Warns about resource usage that may indicate a bug.

```python
import warnings

warnings.warn("This is a resource warning.", ResourceWarning)
```

36. **`DeprecationWarning`**: Warns about features that are deprecated and will be removed in future versions.

```python
import warnings

warnings.warn("This feature is deprecated.", DeprecationWarning)
```

37. **`PendingDeprecationWarning`**: Warns about features that are not deprecated but will be deprecated in the future.

```python
import warnings

warnings.warn("This feature is pending deprecation.", PendingDeprecationWarning)
```

38. **`BytesWarning`**: Issued when mixing bytes and str objects, or comparing bytes and str objects.

```python
import warnings

warnings.warn("Mixing bytes and str is discouraged.", BytesWarning)
```

39. **`UserWarning`**: Warns about user-defined issues.

```python
import warnings

warnings.warn("This is a user-defined warning.", UserWarning)
```

40. **`FutureWarning`**: Issued for warnings about constructs that will change semantically in the future.

```python
import warnings
```

```
warnings.warn("This behavior will change in the future.", FutureWarning)
```

41. **`RuntimeWarning`**: Issued for runtime warnings.

```python
import warnings

warnings.warn("This is a runtime warning.", RuntimeWarning)
```

42. **`SyntaxWarning`**: Issued for warnings about dubious syntax.

```python
import warnings

warnings.warn("This syntax is considered dubious.", SyntaxWarning)
```

43. **`ImportWarning`**: Issued when an import statement triggers a warning.

```python
import warnings

warnings.warn("This import statement triggers a warning.", ImportWarning)
```

44. **`UnicodeWarning`**: Issued for Unicode-related warnings.

```python
import warnings

warnings.warn("This is a Unicode warning.", UnicodeWarning)
```